YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters

Ricardo Rocha, Fernando Silva, and Rolando Martins

DCC-FC & LIACC
University of Porto, Portugal
{ricroc,fds,rolando}@ncc.up.pt

Abstract. This paper discusses the design of YapDss, an or-parallel Prolog system for distributed memory parallel machines, such as the Beowulf PC clusters. The system builds on the work of YapOr, an or-parallel system for shared memory machines, and uses the distributed stack splitting binding model to represent computation state and work sharing among the computational workers. A new variant scheme of stack splitting, the diagonal splitting, is proposed and implemented. This scheme includes efficient algorithms to balance work load among computing workers, to determine the bottommost common node between two workers, and to calculate exactly the work load of one worker. An initial evaluation of the system shows that it is able to achieve very good speedups on a Beowulf PC cluster.

Keywords: Parallel Logic Programming, Or-Parallelism, Stack Splitting.

1 Introduction

Prolog is arguably the most popular Logic Programming language used by researchers in the areas of Artificial Intelligence (AI), such as machine learning and natural language processing. For most of the AI applications, performance is a fundamental issue, and therefore the ability to speedup Prolog execution is a relevant research topic. The development of parallel Prolog systems have further contributed to excel performance. These systems exploit implicit parallelism inherent to the language and therefore do not impose extra work to application developers.

There are two main sources of implicit parallelism in logic programs, *or-parallelism* and *and-parallelism*. Or-parallelism arises from the parallel execution of multiple clauses capable of solving a goal, that is from exploring the non-determinism present in logic programs. And-parallelism arises from the parallel execution of multiple subgoals in a clause body. Of interest to us here is the implementation of or-parallelism. One basic problem with implementing or-parallelism is how to represent, efficiently, the multiple bindings for the same variable produced by the parallel execution of the alternative matching clauses. Two of the most prominent binding models that have been proposed, *binding arrays* and *environment copying*, have been efficiently used in the implementation of Or-Parallel Prolog systems on, mostly, *shared memory platforms* (SMP) [10, 1, 12, 13]. Other proposals have also been put forward addressing the other major type of parallel architectures - *distributed memory platforms* (DMP), also known as *massively parallel processors* [3, 15, 16].

In this paper we are concerned with the implementation of an Or-Parallel Prolog system, the YapDss, for a new type of DMP, namely Beowulf PC clusters. These systems are built from off-the-shelf components and have turned into a viable high-performance, low-cost, scalable, and standardized alternative to the traditional parallel architectures. We take an approach similar to PALS [16], that is we use stack splitting [8] as the main technique to exploit or-parallelism. Stack splitting is a refined version of the environment copying model that is particularly suited for distributed architectures. Environment copying allows computational agents (or workers, or engines, or processors or processes) to share work by copying the state of one busy worker (with unexplored work) to another idle worker. This operation requires further synchronization among the workers to avoid redundant work. Stack splitting introduces a heuristic that when sharing work, the sharing worker completely divides its remaining work with the requesting worker. The splitting is in such a way that both workers will proceed, each executing its branch of the computation, without any need for further synchronization.

Substantial differences between YapDss and PALS resulted in several contributions from our design. The YapDss system builds from a previous efficient Or-Parallel Prolog system, the YapOr [12], based on the environment copying model and the Yap Prolog compiler [14]. YapDss implements a variant stack splitting scheme, the *diagonal splitting*, different from PALS's vertical splitting scheme [16], which, in our view achieves a better work load balance among the computing workers. It uses a simple, yet very efficient, scheme to determine the bottommost common node between the branches of two workers. The work load of a worker is calculated exactly; it is not an estimate. YapDss implements a number of scheduling strategies without having to resort to explicit messages to propagate system work load to workers. Performance analysis showed that YapDss is able to achieve very good performance on a number of common benchmark programs.

The remainder of the paper is organized as follows. First, we introduce the general concepts of environment copying and stack splitting. Next, we describe the diagonal splitting scheme and discuss the major implementation issues in YapDss. We then present an initial performance analysis for a common set of benchmarks. Last, we advance some conclusions and further work.

2 The Multiple Environments Representation Problem

Intuitively, or-parallelism seems simple to implement as the various alternative branches of the search tree are independent of each other. However, parallel execution can result in several conflicting bindings for shared variables. The environments of alternative branches have to be organized in such a way that conflicting bindings can be easily discernible. A binding of a variable is said to be *conditional* if the variable was created before the last choice point, otherwise it is said *unconditional*. The main problem in the management of multiple environments is that of efficiently representing and accessing conditional bindings, since unconditional bindings can be treated as in normal sequential execution.

2.1 Environment Copying

Essentially, the multiple binding representation problem is solved by devising a mechanism where each branch has some private area where it stores its conditional bindings. A number of approaches have been proposed to tackle this problem [7]. Arguably, *environment copying* is the most efficient way to maintain or-parallel environments. Copying was made popular by the Muse or-parallel system [1], a system derived from an early release of SICStus Prolog. Muse showed excellent performance results [2] and in contrast to other approaches, it also showed low overhead over the corresponding sequential system. Most modern parallel logic programming systems, including SICStus Prolog [6], ECLiPSe [17], and Yap [12] use copying as a solution to the multiple bindings problem.

In the environment copying model each worker maintains its own copy of the environment, but in an *identical address space*, that is, each worker allocates their data areas starting at the same logical addresses. An idle worker gets work from a busy worker, by copying all the stacks from the sharing worker. Copying of stacks is made efficient through the technique of *incremental copying*. The idea of incremental copying is based on the fact that the idle worker could have already traversed a part of the search tree that is common to the sharing worker, and thus it does not need to copy this part of stacks. Furthermore, copying of stacks is done from the logical addresses of the sharing worker to exactly the same logical addresses of the idle worker, which therefore avoids potential relocation of address values.

As a result of copying, each worker can carry out execution exactly like a sequential system, requiring very little synchronization with other workers. When a variable is bound, the binding is stored in the private environment of the worker doing the binding, without causing binding conflicts. Synchronization is only needed to guarantee that no two workers explore the same alternative from a shared choice point. Each shared choice point is thus associated with a new data structure, the *shared frame*, that is used to guarantee mutual exclusion when accessing the untried alternatives in such choice points. This works well on SMP, where mutual exclusion is implemented using *locks*. However, mutual exclusion for shared data structures on DMP leads to frequent exchange of messages, which can be a considerable source of overhead and bottleneck.

Nevertheless the shared nature of choice points, environment copying has been recognized as one of the best approaches to support or-parallelism in DMP platforms [5, 4, 3]. This is because, at least all the other data structures, such as the environment, the heap and the trail do not require synchronization. Moreover, practice has showed that the best policy to dispatch work for or-parallel execution is *scheduling on bottommost choice points*. The bottommost policy turns public the whole private region of a worker when it shares work. This maximizes the amount of shared work and possibly avoids that the requesting worker runs out of work too early and therefore invokes the scheduler too often. This is especially important for an environment copying approach because it minimizes the potential number of copying operations.

2.2 Stack Splitting

In order to avoid shared frames, while keeping stack copying with scheduling on bottommost choice points, Gupta and Pontelli proposed a novel technique, called *stack splitting* [8], to ensure that no two workers can pick the same alternative from shared choice points. The basic idea is to split untried alternatives between the workers sharing a choice point. The splitting should be in such a way that each private copy of the choice point in a worker's environment has its own untried alternatives. Several schemes for splitting the set of untried alternatives in shared choice points can be adopted. Figure 1 illustrates three different splitting schemes, that we name *horizontal*, *vertical* and *diagonal* splitting. Other schemes are still possible.

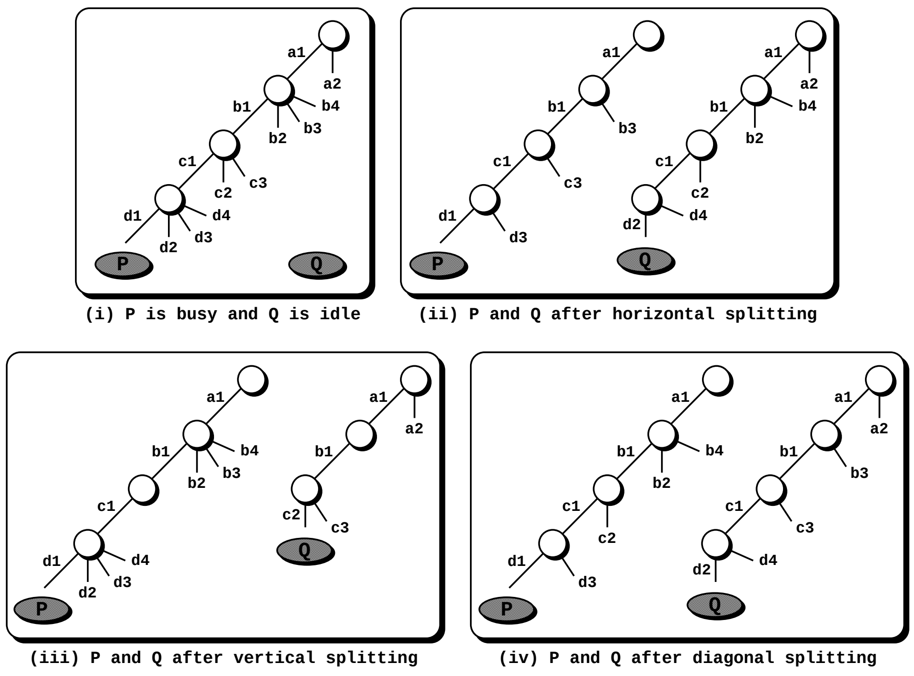
a1
a2
b1
b2
b3
b4
c1
c2
c3
d1
d2
d3
d4
P
Q
(i) P is busy and Q is idle
(ii) P and Q after horizontal splitting
(iii) P and Q after vertical splitting
(iv) P and Q after diagonal splitting

Fig. 1. Splitting of choice points

In horizontal splitting, the untried alternatives in each choice point are alternatively split between the requesting worker Q and the sharing worker P. In vertical splitting, each worker is given all the untried alternatives in alternate choice points, starting from worker P with its current choice point. By observing the figure, it is clear that horizontal and vertical splitting may lead to unbalanced partitioning of the set of untried alternatives between the workers. Despite this fact, the PALS system [16] showed good results by adopting vertical splitting to implement or-parallelism in DMP.

Diagonal splitting uses a more elaborated scheme to achieve a precise partitioning of the set of untried alternatives. It is a kind of mixed approach between horizontal and vertical splitting, the set of untried alternatives in all choice points are alternatively split between both workers. When a first choice point with odd number of untried alternatives (say $2n+1$) appears, one worker (say Q, the one that starts the partitioning) is given $n+1$ alternatives and the other (say P) is given n. The workers then alternate and, in the upper choice point, P starts the partitioning. When more choice points with an odd number of untried alternatives appear, the split process is repeated. At the end, Q and P may have the same number of untried alternatives or, in the worst case, Q may have one more alternative than P.

As a result of applying stack splitting, synchronization through shared frames disappears, environments become completely independent of each other and workers can execute exactly like sequential systems. Workers only communicate when sharing work or when detecting termination. This makes stack splitting highly suitable for distributed execution.

3 The YapDss System

YapDss is an or-parallel Prolog system that implements stack splitting to exploit or-parallelism in DMP. As previous systems, YapDss uses a multi-sequential approach [10] to represent computation state and work sharing among the computational workers. Distributed execution of a program is performed by a set of workers, that are expected to spend most of their time performing useful work. When they have no more alternatives to try, workers search for work from fellow workers. YapDss uses a bottommost policy to dispatch work for or-parallel execution. Work is shared through copying of the execution stacks and diagonal splitting is used to split the available work. Communication among the workers is done using explicit message passing via the LAM implementation [9] of the MPI standard. Our initial implementation does not support cuts or side-effects.

3.1 Splitting Work

A fundamental task when sharing work is to decide which untried alternatives each worker is assigned to. As we have already mentioned, YapDss uses the diagonal scheme to split work among workers. In order to implement that scheme and thus avoid the execution of possible duplicate alternatives, we extended choice points with an extra field, the `CP_OFFSET`. This field marks the offset of the next untried alternative belonging to the choice point. When allocating a choice point, CP_OFFSET is initialized with a value of 1, meaning that the next alternative to be taken is the next alternative in the list of untried alternatives. This is the usual behavior that we expect for private choice points.

With this mechanism, we can easily implement the splitting process when sharing work. We simply need to double the value of the CP_OFFSET field of each shared choice point. This corresponds to alternatively split the set of previous available alternatives in the choice point. To better understand this mechanism, we next illustrate in Fig. 2 a situation where a worker P shares a private choice point with two different workers, first with worker X and later with worker Y.

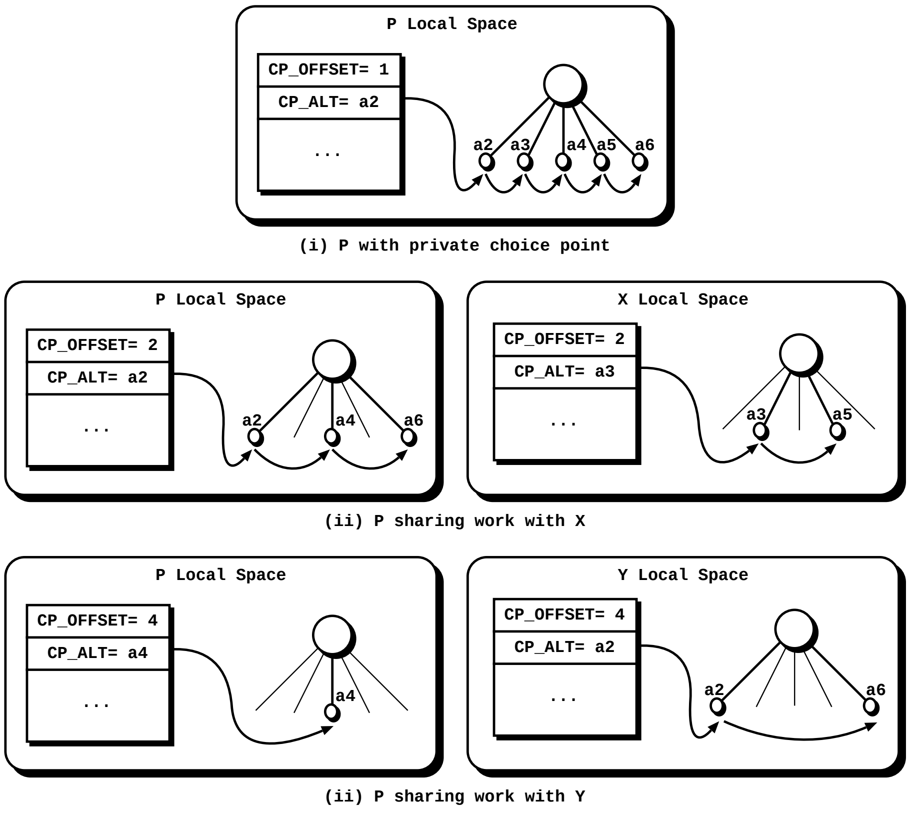

Fig. 2. Using an offset to split work

Initially, in Fig. 2(i), we have P with a private choice point with five untried alternatives: $a2$, $a3$, $a4$, $a5$ and $a6$. CP_OFFSET is 1 and CP_ALT, which holds the reference to the next untried alternative to be taken, refers $a2$. If backtracking occurs, P will successively try each of the five available alternatives. After loading an alternative for execution, P updates CP_ALT to refer to the next one. Because CP_OFFSET is 1, the next alternative to be taken is the next one in the list.

Moving to Fig. 2(ii), consider that P shares its private choice point with X. This can be done by doubling the value in the CP_OFFSET field of the choice point. Moreover, to avoid that both workers execute the same alternatives, the worker that do not start the partitioning of alternatives (please refer to the previous section), X in the figure, updates the CP_ALT field of its choice point to refer to the next available alternative. With this scenario, when backtracking, P will take alternatives $a2$, $a4$ and $a6$ and X will take alternatives $a3$ and $a5$. This happens because they will use the offset 2 to calculate the next reference to be stored in CP_ALT. Finally, in Fig. 2(iii), P shares the choice point with another worker, Y. The value in CP_OFFSET is doubled again and P is the worker that updates the CP_ALT field of its choice point. Within this new scenario, when backtracking, P will take alternative $a4$ and Y will take alternatives $a2$ and $a6$.

When sharing work, we need to know if the number of available alternatives in a choice point is odd or even in order to decide which worker starts the partitioning in the upper choice point. Note that this is not a problem for horizontal or vertical splitting because that kind of information is not needed. A possibility is to follow the list of available alternatives and count its number, but obviously this is not an efficient mechanism. YapDss uses a different approach, it takes advantage of the compiler. All the first instructions that represent the WAM compiled code of a given alternative were extended to include two extra fields in a common predefined position. We will use the names REM_ALT and NEXT_ALT to refer to these fields. Figure 3 shows an example for a predicate with four alternatives: *alt*_1, *alt*_2, *alt*_3 and *alt*_4.

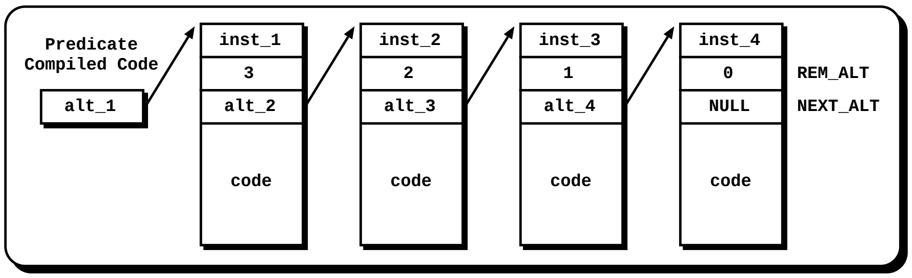

Fig. 3. Compiled code for a predicate in YapDss

The REM_ALT field gives the number of remaining alternatives starting from the current alternative. As we will see next, this allows us to solve the problem of deciding which worker starts the partitioning in a shared choice point. This field was inherited from YapOr.

The NEXT_ALT field is an explicit reference to the compiled code of the next alternative. Note that most of the first instructions that represent the WAM compiled code of a given alternative already contain a reference to the compiled code of the next alternative. The problem is that such references are not positioned in a common predefined position for all instructions. Thus, for these instructions, instead of introducing an extra field we simply make the position uniform. This is extremely important because, when updating the CP_ALT field in a shared choice point (see Fig. 4), we can navigate through the alternatives by simply using the NEXT_ALT field, and therefore avoid testing the kind of instructions they hold to correctly follow the reference to the next alternative.

```
update_alternative(choice point CP) {
  offset = CP_OFFSET(CP)
  next_alt = CP_ALT(CP)
  if (offset > REM_ALT(next_alt))
    next_alt = NULL
  else
    while (offset--)
      next_alt = NEXT_ALT(next_alt)
  CP_ALT(CP) = next_alt
}
```

Fig. 4. Pseudo-code for updating the CP_ALT field in a choice point

The CP_ALT field is updated when a worker backtracks to a choice point to take the next untried alternative or when a worker splits work during the sharing work process. Figure 5 shows the pseudo-code for diagonal splitting. Note that the two workers involved in a sharing operation execute the splitting procedure.

```
diagonal_splitting(bottom choice point BCP, top choice point TCP) {
  if (requesting worker) starts = TRUE  % the requesting worker
  else starts = FALSE                   % starts the partitioning
  current_cp = BCP
  while (current_cp != TCP) {
    alt = CP_ALT(current_cp)
    if (alt != NULL) {
      offset = CP_OFFSET(current_cp)
      if (starts == FALSE)
        update_alternative(current_cp)
      if ((REM_ALT(alt) / offset) mod 2 == 0)  % workers alternate
        starts = !starts
      CP_OFFSET(current_cp) *= 2
    }
    current_cp = CP_B(current_cp)  % CP_B points to the upper choice point
  }
}
```

Fig. 5. Pseudo-code for splitting work using the diagonal scheme

3.2 Finding the Bottommost Common Node

The main goal of sharing work is to position the workers involved in the operation at the same node of the search tree, leaving them with the same computational state. For an environment copying approach, this is accomplished by copying the execution stacks between workers, which may include transferring large amounts of data. This poses a major overhead to stack copying based systems. In particular, for DMP implementations, it can be even more expensive because copying is done through message passing.

To minimize this source of overhead, Ali and Karlsson devised a technique, called *incremental copying* [1], that enables the receiving worker to keep the part of its state that is consistent with that of the giving worker. Only the differences are copied, which permits to reduce considerably the amount of data transferred between workers. However, to successfully implement incremental copying, we need a mechanism that allows us to quickly find the bottommost common node between two workers. For SMP, this is achieved by using the shared frames to store additional information about the workers sharing a choice point [1]. For DMP, we do not have shared data structures where to store that information. To overcome this limitation, in [16], Villaverde and colleagues devised a labeling mechanism to uniquely identify the original source of each choice point (the worker which created it). By comparing private labels from different workers they detect common choice points.

In YapDss we take a similar but simpler approach. We used a private *branch array* to uniquely represent the position of each worker in the search tree. The depth of a choice point along a branch identifies its offset in the branch array. The alternative taken in a choice point defines its value in the branch array. By comparing the branch array of two workers we can easily find the bottommost common node.

Initially, the branch array is empty. When a new choice point is allocated, the top position of the array is marked with the number corresponding to the first alternative to be executed. We take advantage of the REM_ALT field to number the alternatives. For example, consider allocating a choice point for the predicate in Fig. 3, the branch array will be initialized with 3 (the REM_ALT value of the first alternative). When a worker backtracks, the corresponding entry in the branch array is updated with the new REM_ALT value of the next available alternative. Figure 6 shows an example. For simplicity, it considers that all choice points in the figure correspond to predicates with four alternatives, as illustrated in Fig. 3.

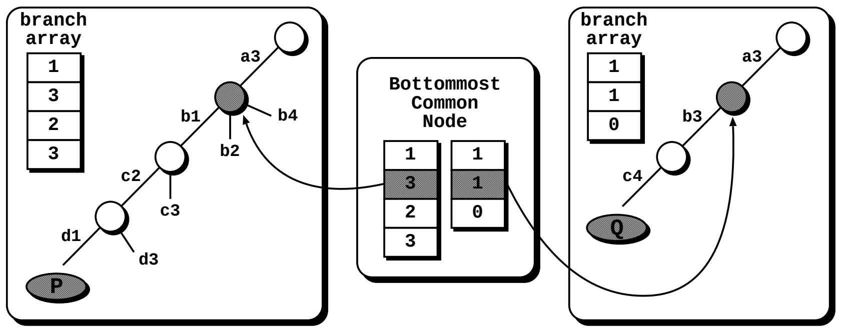

Fig. 6. Using the branch array to find the bottommost common node

Worker P is executing on branch $< a3 : b1 : c2 : d1 >$ and worker Q is executing on branch $< a3 : b3 : c4 >$. Their branch arrays are respectively $< 1 : 3 : 2 : 3 >$ and $< 1 : 1 : 0 >$, which differ in the second entry. We can therefore conclude that the two topmost choice points are common to both workers. They have the same computational state until the bottommost common choice point, and differ bellow such choice point, P is executing alternative $b1$ and Q is executing alternative $b3$. With the branch array data structure, implementing incremental copying for DMP can now be easily done. Moreover, as it uniquely represents the position of each worker in the search tree, we argue that it has good properties that we can take advantage of to extend YapDss to support cuts and side-effects.

3.3 Sharing Work

The sharing work process takes place when an idle worker Q makes a sharing request to a busy worker P and receives a positive answer. In YapDss, the process is as follows. When requesting work, Q sends a message to P that includes its branch array. If P decides to share work with Q, it compares its branch array against the one received from Q in order to find the bottommost common choice point. P then applies incremental copying to compute the stack segments to be copied to Q. It packs all the information in a message and sends it back to Q. If receiving a positive answer, Q copies the stack segments in the message to the proper space in its execution stacks. Meanwhile, P splits the available alternatives in the choice points shared with Q using diagonal splitting. After finishing copying, Q also performs diagonal splitting. While doing diagonal splitting, both workers also update their branch arrays. As an optimization, P can avoid storing its private choice points in the branch array until sharing them.

Note that to fully synchronize the computational state between the two workers, worker Q further needs to install from P the conditional bindings made to variables belonging to the common segments. To solve that, when packing the answering message, P also includes a buffer with all these conditional variables along with their bindings so that Q can update them.

Another point of interest, is how the receiving worker Q can obtain access to the untried alternatives in the choice points of P that are common to both workers [16]. Consider, for example, the situation in Fig. 6. Assuming that P shares work with Q, it will send to Q the stack segments corresponding to its current branch starting from the bottommost common node, that is, branch $< b1 : c2 : d1 >$. Therefore, Q will not be able to access the available alternatives $b2$ and $b4$ in the bottommost choice point because the `CP_ALT` field in its choice point is `NULL`. This can be solved by having P to include in the answering message all the `CP_ALT` fields with available alternatives between the bottommost choice point and the root node [16]. YapDss still does not supports this optimization. Currently, to share untried alternatives from common choice points, P has to explicitly include such segments in the answering message as if they were not common.

3.4 Scheduling

The scheduler is the system component that is responsible for distributing the available work between the various workers. The scheduler must arrange the workers in the search tree in such a way that the total execution time will be the least possible. The scheduler must also minimize the overheads present in synchronization and communication operations such as requesting work, sharing nodes, copying parts of the stacks, splitting work and detecting termination.

An optimal strategy would be to select the busy worker that simultaneously holds the highest work load and that is nearest to the idle worker. The *work load* is a measure of the amount of untried alternatives. *Being near* corresponds to the closest position in the search tree. This strategy maximizes the amount of shared work and minimizes the stacks parts to be copied. Nevertheless, selecting such a worker requires having precise information about the position and work load of all workers. For a DMP based system, maintaining this information requires considerable communications during execution. We thus have a contradiction, to minimize overheads we need more communications. One reasonable solution is to find a compromise between the scheduler efficiency and its overheads. We use a simple but effective strategy to implement scheduling in YapDss.

Each worker holds a private *load register*, as a measure of the exact number of private untried alternatives in its current branch. To compute this exact number we take again advantage of the `CP_OFFSET` and `REM_ALT` fields. The load register is updated in three different situations: when allocating a new choice point, it is incremented by the number of untried alternatives left in the choice point; when backtracking, it is decremented by one unit; and when splitting work, it can be incremented (if receiving work) or decremented (if giving work) by the number of alternatives being split.

Besides, each worker holds a private *load vector* as a measure of the estimated work load of each fellow worker. The load vector is updated in two situations: when sharing work and when receiving a termination token. YapDss does not introduce specific messages to explicitly ask for work load information from a worker, and instead it extends the existing messages to include that information. A trivial case occurs when a worker receives a sharing request, a zero work load can be automatically inferred for the requesting worker. When sharing work, the answering message is extended to include the work load of the giving worker. When detecting termination, the termination tokens are extended to include the work load of all workers in the system.

Termination detection is done using a simple algorithm from [11]. When an idle worker Q suspects that all the other workers are idle too, it initializes a termination token with its work load (zero in this case) and sends it to the next worker on rank. When receiving a token, a worker updates its load vector with the load information already in the token, includes its work load and sends it to the next worker on rank. The process repeats until reaching the initial worker Q. If, when reaching Q, the token is clean (zero load for all workers) then Q broadcasts a termination message. Otherwise, Q simply updates its load vector and starts scheduling for a busy worker.

When scheduling for a busy worker, we follow the following strategies. By default, an idle worker Q tries to request work from the last worker, say L, which has shared work with it in order to minimize the potential stacks parts to be copied. However, if the work load for L in Q's load vector is less than a threshold value `LOAD_BALANCE` (6 in our implementation), a different strategy is used, Q starts searching its load vector for the worker with the greatest work load to request work from it (note that L can be the selected worker).

When a worker P receives a sharing request, it may accept or refuse the request. If its current work load is less than `LOAD_BALANCE`, it replies with a negative answer. Otherwise, it accepts the sharing request and, by default, performs stack splitting until the bottommost common node N. However, if the available work in the branch until node N is less than `LOAD_BALANCE`, it may extend the splitting branch to include common choice points (please refer to the last paragraph in section 3.3). In both cases (negative or positive answers), P includes in the answering message its current work load. If receiving a negative answer, Q updates its load vector with the value for P and starts searching for the next worker with the greatest work load. Meanwhile, if Q finds that all entries in its load vector are zero, it initializes a termination token.

4 Initial Performance Evaluation

The evaluation of the first implementation of YapDss was performed on a low-cost PC cluster with 4 dual Pentium II nodes interconnected by Myrinet-SAN switches. The benchmark programs are standard and commonly used to assess other parallel Prolog systems. All benchmarks find all solutions for the problem. We measured the timings and speedups for each benchmark and analyzed some parallel activities to identify potential sources of overhead.

To put the performance results in perspective we first evaluate how YapDss compares against the Yap Prolog engine. Table 1 shows the base running times, in seconds, for Yap and YapDss (configured with one worker) for the set of benchmark programs. In parentheses, it shows YapDss's overhead over Yap running times. The results indicate that YapDss is on average 16% slower than Yap. YapDss overheads mainly result from handling the work load register, the branch array, and from testing operations that check for termination tokens or sharing request messages.

Table 2 presents the performance of YapDss with multiple workers. It shows the running times, in seconds, for the set of benchmark programs, with speedups relative to the one worker case given in parentheses. The running times correspond to the best times obtained in a set of 5 runs. The variation between runs was not significant.

The results show that YapDss is quite efficient in exploiting or-parallelism, giving effective speedups over execution with just one worker. The quality of the speedups achieved depends significantly on the amount of parallelism in the program being executed. The programs in the first group, *queens12*, *nsort*, *puzzle4x4* and *magic*, have rather large search spaces, and are therefore amenable

Programs	Yap	YapDss
nsort	188.50	218.68(1.16)
queens12	65.03	72.80(1.12)
puzzle4x4	54.61	67.91(1.24)
magic	29.31	30.90(1.05)
cubes7	1.26	1.31(1.05)
ham	0.23	0.30(1.32)
Average		(1.16)

Table 1. Running times for Yap and YapDss with one worker

	Number of Workers			
Programs	**2**	**4**	**6**	**8**
queens12	38.93(1.99)	19.63(3.94)	13.36(5.80)	10.12(7.66)
nsort	124.24(1.98)	63.14(3.90)	42.44(5.80)	33.06(7.45)
puzzle4x4	34.00(1.99)	17.34(3.91)	11.83(5.73)	9.41(7.20)
magic	15.50(1.99)	7.88(3.92)	5.58(5.53)	4.38(7.05)
cubes7	0.67(1.96)	0.40(3.26)	0.33(3.90)	0.23(4.80)
ham	0.17(1.75)	0.10(2.81)	0.09(3.13)	0.10(2.95)
Average	(1.94)	(3.62)	(4.98)	(6.19)

Table 2. Running times and speedups for YapDss with multiple workers

to the execution of coarse-grained tasks. This group shows very good speedups up to 8 workers. The speedups are still reasonably good for the second group, programs *cubes7* and *ham*, given that they have smaller grain tasks.

We next examine the main activities that take place during parallel execution in order to determine which of them are causing a decrease in performance. The activities traced are:

Prolog: percentage of total running time spent in Prolog execution and in handling the work load register and branch array.
Search: percentage of total running time spent searching for a busy worker and in processing sharing and termination messages.
Sharing: percentage of total running time spent in the sharing work process.
Reqs Acp: total number of sharing request messages accepted.
Reqs Ref: total number of sharing request messages refused.
Recv Load: total number of untried alternatives received from busy workers during splitting.

Table 3 shows the results obtained with 2, 4 and 8 workers in each activity for two of the benchmark programs, one from each class of parallelism.

The results show that when we increase the number of workers, the percentage of total running time spent on the *Prolog* activity tends to decrease and be moved to the *Search* activity. This happens because the competition for finding work leads workers to get smaller tasks. This can be observed by the increase

	Activities					
Programs	Prolog	Search	Sharing	Reqs Acp	Reqs Ref	Recv Load
queens12						
2 workers	99%	0%	1%	8	3	296
4 workers	98%	1%	1%	99	208	3568
8 workers	93%	6%	1%	160	6630	5592
ham						
2 workers	80%	19%	1%	6	3	64
4 workers	53%	45%	2%	15	87	135
8 workers	25%	74%	1%	18	346	195

Table 3. Workers activities during execution

in the *Recv Load* parameter. If workers get smaller tasks, they tend to search for work more frequently. This can be observed by the increase in the *Reqs Acp* and *Reqs Ref* parameters. The time spent in the *Sharing* activity is almost constant, suggesting that the splitting process is not a major problem for YapDss performance.

5 Concluding Remarks

In this paper we proposed a new variant scheme of the stack splitting scheme, the diagonal splitting, and described its implementation in the YapDss or-parallel Prolog system. This scheme includes efficient algorithms to balance work load among computing workers, to determine the bottommost common node between two workers, and to calculate exactly the work load of one worker.

YapDss showed good sequential and parallel performance on a set of standard benchmark programs, running on a PC cluster parallel architecture. It was able to achieve excellent speedups for applications with coarse-grained parallelism and quite good results globally. This may be a result of the low communication overheads imposed by the scheduling schemes implemented.

Future work include more detailed system evaluation and performance tuning, in particular we intend to evaluate the system on a recently built PC cluster with 4 dual AMD XP 2000+ nodes, with 2 GBytes of main memory per node, interconnected by Giga-Ethernet switches. We also plan to extend YapDss to better support all builtins, support speculative execution with cuts, and integrate the system in the Yap distribution.

Acknowledgments

We are grateful for the insightful comments received from Vítor Santos Costa. This work has been partially supported by APRIL (POSI/SRI/40749/2001), and by funds granted to LIACC through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI.*

References

1. K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.
2. K. Ali, R. Karlsson, and S. Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. *New Generation Computing*, 11(1 & 4):81–103, 1992.
3. L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, 1997.
4. V. Benjumea and J. M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, number 714 in Lecture Notes in Computer Science, pages 291–301. Springer-Verlag, 1993.
5. J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. OPERA: Or-Parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, pages 45–64. Wiley & Sons, New York, USA, 1992.
6. M. Carlsson and J. Widen. SICStus Prolog User's Manual. SICS Research Report R88007B, Swedish Institute of Computer Science, 1988.
7. G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages*, 15(4):659–680, 1993.
8. G. Gupta and E. Pontelli. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *Proceedings of the 16th International Conference on Logic Programming*, pages 290–304. The MIT Press, 1999.
9. Open Systems Laboratory. LAM/MPI Parallel Computing, 2003. Available from `http://www.lam-mpi.org`.
10. E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, 1988.
11. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.
12. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, number 1695 in Lecture Notes in Artificial Intelligence, pages 178–192. Springer-Verlag, 1999.
13. R. Rocha, F. Silva, and V. Santos Costa. On a Tabling Engine that Can Exploit Or-Parallelism. In *Proceedings of the 17th International Conference on Logic Programming*, number 2237 in Lecture Notes in Computer Science, pages 43–58. Springer-Verlag, 2001.
14. V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 261–267, Paris, France, 1999. Springer-Verlag.
15. F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 43(2):173–186, 2000.
16. K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *Proceedings of the 17th International Conference on Logic Programming*, number 2237 in Lecture Notes in Computer Science, pages 27–42. Springer-Verlag, 2001.
17. M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. Technical report, IC-Parc, Imperial College, London, 1997.