Or-Parallel Prolog Execution on Clusters of Multicores

João Santos and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{jsantos,ricroc}@dcc.fc.up.pt

Abstract

Logic Programming languages, such as Prolog, provide an excellent framework for the parallel execution of logic programs. In particular, the inherent non-determinism in the way logic programs are structured makes Prolog very attractive for the exploitation of *implicit parallelism*. One of the most noticeable sources of implicit parallelism in Prolog programs is *or-parallelism*. Or-parallelism arises from the simultaneous evaluation of a subgoal call against the clauses that match that call. Arguably, the most successful model for or-parallelism is *environment copying*, that has been efficiently used in the implementation of or-parallel Prolog systems both on shared memory and distributed memory architectures. Nowadays, multicores and clusters of multicores are becoming the norm and, although, many parallel Prolog systems have been developed in the past, to the best of our knowledge, none of them was specially designed to explore the combination of shared with distributed memory architectures. Motivated by our past experience, in designing and developing parallel Prolog systems based on environment copying, we propose a novel computational model to efficiently exploit implicit parallelism from large scale real-world applications specialized for the novel architectures based on clusters of multicores.

1998 ACM Subject Classification D.1.3 Concurrent Programming

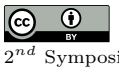
Keywords and phrases Logic Programming, Or-Parallelism, Environment Copying, Scheduling

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.9

1 Introduction

Logic Programming languages, such as Prolog, provide a high-level, declarative approach to programming. In general, logic programs can be seen as executable specifications that despite their simple declarative and procedural semantics allow for designing very complex and efficient applications. The inherent non-determinism in the way logic programs are structured as simple collections of alternative logic clauses makes Prolog very attractive for the exploitation of *implicit parallelism*.

Prolog offers two major forms of implicit parallelism: *and-parallelism* and *or-parallelism* [5]. And-Parallelism stems from the parallel evaluation of subgoals in a clause, while or-parallelism results from the parallel evaluation of a subgoal call against the clauses that match that call. Arguably, or-parallel systems, such as Aurora [7] and Muse [3], have been the most successful parallel logic programming systems so far. Intuitively, the least complexity of or-parallelism makes it more attractive as a first step. However, practice has shown that a main difficulty, when implementing or-parallelism, is how to efficiently represent the *multiple bindings* for the same variable produced by the parallel execution of alternative matching clauses. One of the most successful or-parallel models that solves the multiple bindings problem is *environment copying*, that has been efficiently used in the



2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 9–20

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implementation of or-parallel Prolog systems both on shared memory [3, 10] and distributed memory [16, 9] architectures.

Another major difficulty in the implementation of any parallel system is the design of *scheduling strategies* to efficiently assign computing tasks to idle workers. A parallel Prolog system is no exception as the parallelism that Prolog programs exhibit is usually highly irregular. Achieving the necessary cooperation, synchronization and concurrent access to shared data among several workers during execution is a difficult task. For environment copying, scheduling strategies based on *dynamic scheduling* of work have proved to be very efficient [2]. *Stack splitting* [4, 8] is an alternative scheduling strategy for environment copying that provides a simple and clean method to accomplish work splitting among workers in which all available work is *statically divided beforehand* in complementary sets between the sharing workers. Due to its static nature, stack splitting was thus first introduced aiming at distributed memory architectures [16, 9] but, recent work, also showed good results for shared memory architectures [15, 14].

The increasing availability and popularity of multicore processors have made our personal computers parallel with multiple cores sharing the main memory. Multicores and clusters of multicores are now the norm and, although, many parallel Prolog systems have been developed in the past, most of them are no longer available, maintained or supported. Moreover, to the best of our knowledge, none of them was specially designed to explore the combination of shared with distributed memory architectures. On one hand, the shared memory based models take advantage of synchronization mechanisms that cannot be easily extended to distributed environments and, on the other hand, the distributed memory based models use specialized communication mechanisms that do not take advantage of the fact that some workers can be sharing memory resources.

Motivated by the intrinsic and strong potential that Prolog has for implicit parallelism and by our past experience in designing and developing parallel systems based on environment copying [10, 9, 15, 14], we propose a novel computational model to efficiently exploit parallelism from large scale real-world applications specialized for clusters of low cost multicore architectures. In this new model, we will have two levels of computational units, *single workers* and *teams of workers*, and the ability to exploit different scheduling strategies, for distributing work among teams and among the workers inside a team. Our approach resembles the concept of teams used by some of the models combining and-parallelism with or-parallelism, like the Andorra-I [13] or ACE [6] systems, where a layered approach implements different schedulers to deal with each level of parallelism.

In our model, a team of workers is formed by workers sharing the same memory address space, i.e., two workers executing in different computer nodes cannot belong to the same team, but we can have more than a team executing in the same computer node. For (shared memory) multicores, we can thus have any combination of strategies, teams and workers inside a team can distribute work using both dynamic or static scheduling of work. For (distributed memory) clusters of multicores, we can only have (static) stack splitting for distributing work among teams, but we can still have dynamic or static scheduling of work for distributing work among the workers inside a team. This idea is similar to the MPI/OpenMP hybrid programming pattern, where MPI is usually used to communicate work among workers in different computer nodes and OpenMP is used to communicate work among workers in the same node.

The remainder of the paper is organized as follows. First, we introduce some background about environment copying, stack splitting and work scheduling. Next, we introduce our new model and discuss the major design issues, algorithms and challenges. Last, we ad-

vance directions for further work. Throughout the text, we assume the reader will have good familiarity with the general principles of Prolog implementation, and namely with the WAM [18, 1]. When discussing some technical details, we will take as reference the state-of-the-art Yap Prolog system [12], that integrates or-parallelism based on the environment copying model and supports both dynamic and static scheduling of work.

2 Environment Copying

In the environment copying model, each worker keeps a separate copy of its own environment, thus the bindings to shared variables are done as usual (i.e., stored in the private execution stacks of the worker doing the binding) and without conflicts. Every time a worker shares work with another worker, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to the search tree node where the sharing occurs. At the engine level, a search tree node corresponds to a choice point in the local stack [18, 1].

As a result of environment copying, each worker can proceed with the execution exactly as a sequential engine, with just minimal synchronization with other workers. Synchronization is mostly needed when updating scheduling data and when accessing shared nodes in order to ensure that unexplored alternatives are only exploited by one worker. All other WAM data structures, such as the environment frames, the heap, and the trail do not require synchronization.

2.1 Incremental Copying

To reduce the overhead of stack copying, an optimized copy mechanism called *incremental copy* [3] takes advantage of the fact that the requesting worker may already have traversed part of the path being shared. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest node common to both workers.

For example, consider that worker Q asks worker P for sharing and that worker P decides to share its private nodes with Q. To implement incremental copying, Q should start by backtracking to the youngest common node with P, therefore becoming partially consistent with part of P. Then, if Q receives a positive answer from P, it only needs to copy the differences between P and Q. These differences can be easily calculated through the information stored in the common node found by Q and in the top registers of the execution stacks of P. Care must be taken about variables older than the youngest common node that were instantiated by P, as incremental copying does not copy these bindings. Worker Q thus needs to explicitly *install* the bindings for such variables. This process, called the *adjustment of cells outside the increments*, is implemented by searching the trail stack for bindings to variables older than the youngest common node [3].

2.2 Or-Frames

Deciding which workers to ask for work and how much work should be shared is a function of the scheduler. A fundamental task when sharing work is to turn *public* the private choice points, so that backtracking to these choice points can be synchronized between different workers. Public choice points are treated differently because we need to synchronize workers in such a way that we avoid executing twice the same alternative.

Strategies based on dynamic scheduling of work, use *or-frames* to implement such synchronization [3]. A worker sharing work adds an or-frame data structure to each private choice point made public. Each or-frame stores the pointer to the next available alternative, as previously stored in the corresponding private choice point, and supports a mutual exclusion mechanism that guarantees atomic updates to the or-frame data. Shared nodes thus become represented by or-frames, a data structure that workers must access, with mutual exclusion, to obtain the unexplored alternatives. The set of all or-frames form a tree that represents the public search tree.

2.3 Stack Splitting

Stack splitting was first introduced to target distributed memory architectures, thus aiming to reduce the mutual exclusion requirements of the or-frames when accessing shared nodes of the search tree. It accomplishes this by defining simple and clean work splitting strategies in which all available work is statically divided beforehand in two complementary sets between the sharing workers. In practice, with stack splitting the synchronization requirement is removed by the preemptive split of all unexplored alternatives at the moment of sharing. The splitting is such that both workers will proceed, each executing its branch of the computation, without any need for further synchronization when accessing shared nodes.

The original stack splitting proposal [4] introduced two strategies for dividing work: *vertical splitting*, in which the available choice points are alternately divided between the two sharing workers, and *horizontal splitting*, which alternately divides the unexplored alternatives in each available choice point. *Diagonal splitting* [9] is a more elaborated strategy that achieves a precise partitioning of the set of unexplored alternatives. It is a kind of mix between horizontal and vertical splitting, where the set of all unexplored alternatives in the available choice points is alternately divided between the two sharing workers. Another splitting strategy [17], which we named *half splitting*, splits the available choice points in two halves. Figure 1 illustrates the effect of these strategies in a work sharing operation between a busy worker P and an idle worker Q.

Figure 1(a) shows the initial configuration with the idle worker Q requesting work from a busy worker P with 7 unexplored alternatives in 4 choice points. Figure 1(b) shows the effect of vertical splitting, in which P keeps its current choice point and alternately divides with Q the remaining choice points up to the root choice point. Figure 1(c) illustrates the effect of half splitting, where the bottom half is for worker P and the half closest to the root is for worker Q. Figure 1(d) details the effect of horizontal splitting, in which the unexplored alternatives in each choice point are alternately split between both workers, with workers P and Q owning the first unexplored alternative in the even and odd choice points, respectively. Figure 1(e) describes the diagonal splitting strategy, where the unexplored alternatives in all choice points are alternately split between both workers in such a way that, in the worst case, Q may stay with one more alternative than P. For all strategies, the corresponding execution stacks are first copied to Q, next both P and Q perform splitting, according to the splitting strategy at hand, and then P and Q are set to continue execution.

2.4 The Yap Prolog System

The Yap Prolog system implements or-parallelism based on the environment copying model and supports both dynamic and static scheduling of work. To implement dynamic scheduling, Yap follows the original Muse approach which uses or-frames to synchronize the access to the open alternatives. To implement static scheduling, two different approaches were

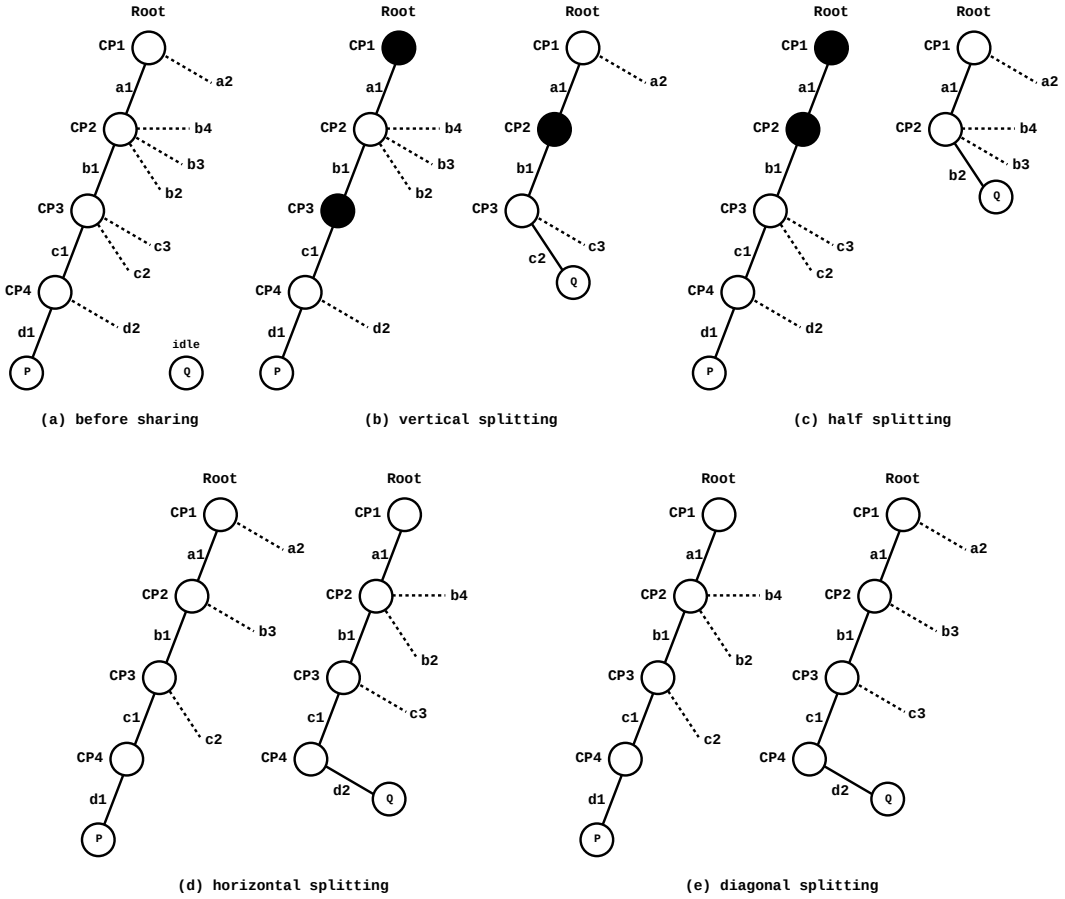


Figure 1 Alternative stack splitting strategies.

followed. In the first approach, the engine was designed to run in Beowulf clusters [9]. More recently, a second approach was designed to run in multicores and it has shown to be very competitive when compared with the original or-frames approach [15, 14].

When running in shared memory architectures, Yap's workers can be either processes (the engine using processes is called YapOr [10]) or POSIX threads (the engine using threads is called ThOr [11]). The memory organization for YapOr/ThOr is quite similar for all the approaches (see Fig. 2(a)). The memory of the system is divided into two major address spaces: the *global space* and a collection of *local spaces*. The global space contains the code area inherited from Yap and all data structures necessary to support parallelism. Among these structures is static information about the execution, such as the number of workers, and dynamic information responsible for determining the end of the execution. Each local space represents one worker and contains the execution stacks inherited form Yap (heap, local, trail and auxiliary stack) and information related to the execution of that worker such as the top shared choice point, share and prune requests or the load of that worker [10, 11].

When running in distributed memory architectures, Yap's workers are processes, each with independent global and local spaces (see Fig. 2(b)). Despite not specially designed for it, this approach also fits in shared memory architectures, i.e., we can have some workers running on the same computer node, but as fully independent processes.

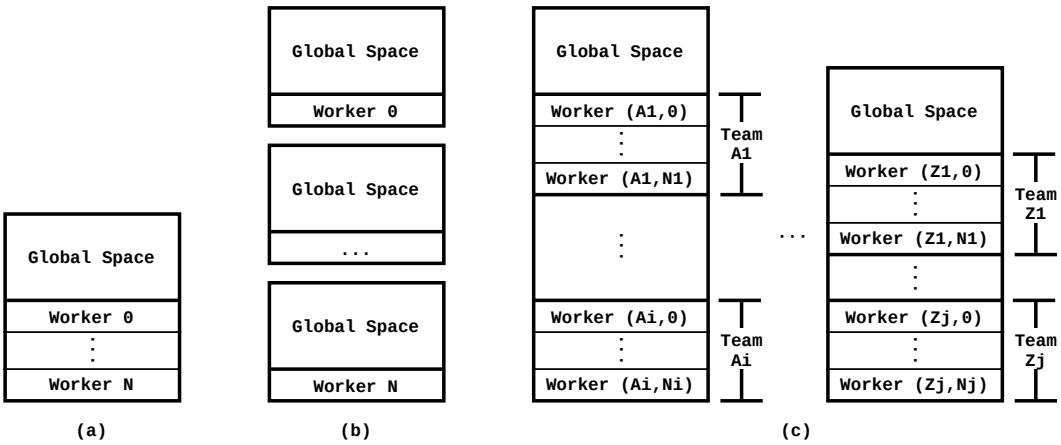


Figure 2 Memory layout for: (a) workers in shared memory; (b) workers in distributed memory; and (c) teams of workers in clusters of multicores.

3 Our Proposal

The goal behind our proposal is to implement the concept of teams trying to reuse, as much as possible, Yap's existing infrastructure. We define a team as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem. By demanding that all workers inside a team share the same address space implies that all workers should be in the same computer node. On the other hand, we also want to be possible to have several teams in a computer node or distributed by other nodes.

3.1 Memory Organization

In order to support teams, there are several changes that need to be made, being one of the first, the memory organization. Figure 2(c) shows the new memory layout to support teams of workers. Each team of workers mimics the previous memory layout for a set of workers in shared memory (see Fig. 2(a)), where the memory of the system is divided into a global space, shared among all workers, and a collection of local spaces, each representing one worker's team. In this new memory layout, we can also have several teams sharing the same memory address space and, in particular, sharing the global space. To accomplish that, the information stored in the global space is now related with teams instead of being related with single workers. Moreover, the global space now includes an extra area, named *team space*, where each team stores static information about the team and dynamic information about the execution of the team, such as, to determine if the team is out of work or if it has finished execution. The collection of local spaces maintains its functionality, i.e., it stores the execution stacks and information about the state of the corresponding worker.

Since our aim is to target clusters of multicores, the complete layout for the new memory organization can be seem as a generalization of the previous approach for distributed memory architectures (see Fig. 2(b)), but now instead of single workers with independent global and local spaces, we may have teams, individual teams or collection of teams as described above, sharing the same memory address space.

3.2 Mixed Scheduling

One of the main advantages of using teams is that we can combine the scheduling strategies mentioned before. Therefore we may have teams using static scheduling while others, at the same time, use dynamic scheduling. Figure 3 shows a schematic representation of what we want to achieve with our proposal. In this example, we have a cluster composed by two computers nodes, $N1$ and $N2$. The computer node $N1$ has two teams, team A and team B with 4 workers each. The computer node $N2$ has only one team, team C with 8 workers.

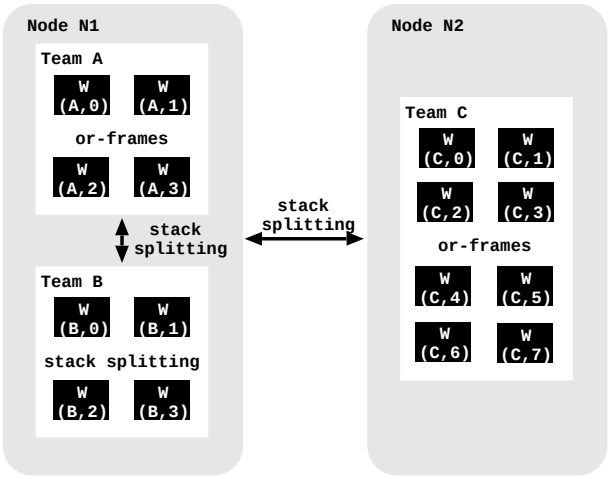


Figure 3 Work scheduling within and among teams.

Regarding the scheduling strategy adopted to distribute work inside the teams, teams A and C are using dynamic scheduling with or-frames, while team B is using stack splitting. To distribute work among teams, we only use stack splitting. This is mandatory since we want to have a single scheduling protocol to distribute work between teams (being they in the same or in different computer nodes) and we want to fully avoid having synchronization data structures, such as the or-frames, being shared between teams. Note that having the access to the open alternatives in data structures shared between teams, not only would have a great impact in the communication overhead required to keep them up-to-date, but would also not clarify the notion of being a team. If two teams are synchronizing the access to the open alternatives, in fact they are not two different teams but only one, because no decision regarding the shared open alternatives can be made without involving both teams.

Independently of the scheduling strategy, teams will have to communicate among them when sharing work or when sending requests to perform a cut or to ensure the termination of the computation. To implement the communication layer, we can use a message passing protocol, for teams physically located in the same or in different computer nodes, or a shared memory synchronization mechanism, for teams in the same computer node. Note that, in this latter case, synchronization is being use to implement communication and not for scheduling purposes, as discussed before.

3.3 Work Sharing

To distribute work inside a team, we can use, with minor adaptations, any of Yap's current dynamic or static schedulers for shared memory. Since these schedulers were developed to deal with workers that are sharing the same memory address space, they can thus be easily

extended to support work sharing inside a team. As discussed before, this is not the case for work sharing among teams. To deal with that, our approach is thus to implement a layered approach, similar to the one used by some of the models combining and-parallelism with or-parallelism [13, 6], and for that a second-level scheduler will be used.

Since the concept of a team implies that we must give priority to the exploitation of the work available inside the team, we will only ask for work to other teams when no more work exists in a team. However, even though that it is the entire team that is out of work, the sharing process will still be done between two workers, being the selected worker of the idle team then the responsible for sharing the new work with its teammates.

Figure 4 shows a schematic representation of the sharing process between teams. Consider the cluster configuration in Fig. 3 and assume that team C has run out of work and that team A was selected by C's scheduler to share work with it. Figure 4(a) shows the state of team A before the sharing request from C. The four workers in team A are executing in the private region of the search tree and all share the top three choice points. The top shared choice point is already dead, i.e., without open alternatives, but the second and third shared choice points have two ($b2$ and $b3$) and one ($c3$) open alternatives, respectively.

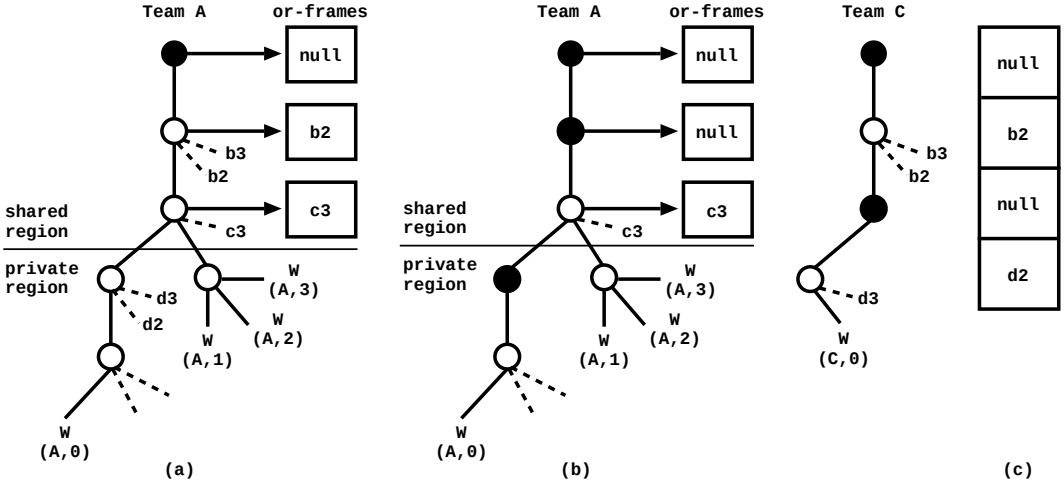


Figure 4 Schematic representation of the sharing process between workers of different teams: in (a) we can see the configuration of team A when team C asks for work and in (b) we can see the configuration of both teams after the sharing process, considering that worker $W(A,0)$ used vertical splitting to share its available work (in (c) we can see the array of open alternatives being shared) with worker $W(C,0)$.

When team A receives the sharing request from team C, one of the workers from A will be selected to share part of its available (private and/or shared) work and manage the sharing process with the requesting worker from C. For the sake of simplicity, here we are considering that this is done by the workers 0 of each team, workers $W(A,0)$ and $W(C,0)$. Since this is a sharing operation between teams, static scheduling is then the strategy adopted to split work. In particular, in this example, we are using the vertical splitting strategy.

To implement vertical splitting, $W(A,0)$ thus needs to alternately divide its choice points with $W(C,0)$. However, since team A is using or-frames to implement dynamic scheduling of work inside the team, we cannot apply the original stack splitting algorithm [15, 14] to split the available work in the shared region of the search tree (please remember that stack splitting avoids the use of or-frames). To solve that problem, $W(A,0)$ constructs an array

with the open alternatives per choice point that it will hand over to $W(C,0)$. This array is illustrated in Fig. 4(c). The motivation for using this array is the isolation between the alternatives being shared and the scheduling strategy being used, therefore allowing that two teams can share work, independently of their scheduling strategies. Note that, when splitting work in a shared choice point, first $W(A,0)$ needs to gain (lock) access to the corresponding or-frame, then it moves the next unexplored alternative from the or-frame to the array of open alternatives, updates the or-frame to *null* and unlocks it.

At the end, the array with the open alternatives and the execution stacks of $W(A,0)$ are copied to $W(C,0)$. Figure 4(b) shows the configuration of both teams after the sharing process. In team A, we can see the effect of vertical splitting by observing the new dead nodes in the branch of $W(A,0)$. In team C, we can see that $W(C,0)$ instantiated the work received from $W(A,0)$ as fully private work. $W(C,0)$ will only share its work, and allocate the corresponding or-frames if team C is also using dynamic scheduling, when the scheduler inside the team notifies it to share work with its teammates.

3.4 Algorithms

In this section, we present in more detail the two algorithms that implement the key aspects of our new model.

Algorithm 1 shows the pseudo-code for the $WorkerGetWork()$ procedure that, given an idle worker W belonging to a team T, searches for a new piece of work for W. In a nutshell, we can resume the algorithm as follows. Initially, W starts by selecting a busy worker B from its teammates to potentially share work with (line 3). Next, it sends a share request to B (line 4) and if the request gets accepted, then both workers perform the work sharing procedure, according to the scheduling strategy (dynamic or static) being used in T (line 5). After sharing, W returns to Prolog execution (line 6). Otherwise, if the sharing request gets refused, then W should try another busy worker from T, while there are teammates with available work (line 2).

Algorithm 1 $WorkerGetWork(W,T)$.

```
1:  while TeamNotFinished(T) do
2:    while TeamWithWork(T) do
3:      B ← SelectBusyWorker(T)
4:      if SendShareRequest(W, B) = ACCEPTED then
5:        ShareWork(W, B)
6:        return true
7:    if W = SelectMasterWorker(T) then {W will search for work from the other teams}
8:      if TeamGetWork(W, T) then {worker W has obtained work from another team}
9:        return true
10:     else {all teams should finish execution}
11:       SetTeamAsFinished(T)
12: return false
```

On the other hand, if all workers in T run out of work (i.e., if all workers are executing the $WorkerGetWork()$ procedure), then one of the workers from T, named the *master worker* W, will be selected to search for work from the other teams (line 7), and for that it executes the $TeamGetWork()$ procedure (line 8), as explained next in Algorithm 2. If the call to $TeamGetWork()$ succeeds, this means that W has obtained a new piece of work from another team and, in such case, W returns to Prolog execution to start exploiting the new

available work (line 9). Otherwise, if the call to $TeamGetWork()$ fails, this means that all teams are out of work and, in such case, team T is set as finished (line 11) and all workers in T then finish execution by returning $false$ (line 12).

Next, Algorithm 2 shows the pseudo-code for the $TeamGetWork()$ procedure that, given the master worker W of an idle team T, searches for a new piece of work from the other teams. Initially, W starts by selecting a busy team U from the available set of teams to potentially share work with (line 2). Next, it sends a share request to team U (line 3) and if the request gets accepted, then W performs the work sharing procedure, with the selected sharing worker S from U (lines 4–5), and returns successfully (line 6). Otherwise, if the sharing request gets refused, then W should try another busy team, while there teams with available work (line 1). On the other hand, if all teams run out of work (i.e., if all master workers are executing the $TeamGetWork()$ procedure), then W returns failure (line 7).

Algorithm 2 $TeamGetWork(W, T)$.

```
1: while not AllTeamsWithoutWork() do
2:   U ← SelectBusyTeam()
3:   if SendShareRequest(T, U) = ACCEPTED then
4:     S ← GetSharingWorker(U)
5:     ShareWork(W, S)
6:     return true
7: return false
```

4 Conclusions

We have proposed a novel computational model to efficiently exploit implicit or-parallelism from large scale real-world applications specialized for the novel architectures based on clusters of multicores. The main goal behind our proposal is to implement the concept of teams in order to decouple the scheduling of work from the architecture of the system. In particular, we are most interested in the ability of exploiting different scheduling strategies for distributing work among workers and among teams in the same or in different computer nodes.

Currently, we have already started the implementation of the new model in the Yap Prolog system, trying to reuse, as much as possible, the existing infrastructure that supports both dynamic and static scheduling of work for or-parallelism based on the environment copying model. Beyond the implementation of the initial prototype, further work will include: (i) studying load balancing, i.e., how to better distribute work across teams and across workers in a team; (ii) avoid speculative work, i.e., avoid work which would not be done in a sequential system; and (iii) support sequential semantics, i.e., predicate side-effects must be executed by leftmost workers, as otherwise we may change the sequential behavior of the program.

Acknowledgments This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects LEAP (FCOMP-01-0124-FEDER-015008) and PEst (FCOMP-01-0124-FEDER-022701). João Santos is funded by the FCT grant SFRH/BD/76307/2011.

References

1. H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.
2. K. Ali and R. Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.
3. K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.
4. G. Gupta and E. Pontelli. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *International Conference on Logic Programming*, pages 290–304. The MIT Press, 1999.
5. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.
6. G. Gupta, E. Pontelli, M. V. Hermenegildo, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–109. The MIT Press, 1994.
7. 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 *International Conference on Fifth Generation Computer Systems*, pages 819–830. Institute for New Generation Computer Technology, 1988.
8. E. Pontelli, K. Villaverde, Hai-Feng Guo, and G. Gupta. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing*, 66(10):1267–1293, 2006.
9. R. Rocha, F. Silva, and R. Martins. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Portuguese Conference on Artificial Intelligence*, number 2902 in LNAI, pages 136–150. Springer-Verlag, 2003.
10. R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.
11. V. Santos Costa, I. Dutra, and R. Rocha. Threads and Or-Parallelism Unified. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 10(4–6):417–432, 2010.
12. V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.
13. V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, 1991.
14. R. Vieira, R. Rocha, and F. Silva. On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 71–85, 2012.
15. R. Vieira, R. Rocha, and F. Silva. Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In *International Workshop on Declarative Aspects and Applications of Multicore Programming*. ACM Digital Library, 2012.
16. K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 27–42. Springer-Verlag, 2001.
17. K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In *International Euro-Par Conference*, number 2790 in LNCS, pages 694–703. Springer-Verlag, 2003.

18 D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.