Computer Science and Information Systems 11(4):1435–1454 DOI: 10.2298/CSIS131025050S

A Team-Based Scheduling Model for Interfacing Or-Parallel Prolog Engines

Joao 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, offer a great potential 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. 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 and distributed memory architectures. Conceptually, an or-parallel Prolog system consists of two components: an or-parallel engine (i.e., a set of independent Prolog engines which we named a team of workers) and a scheduler. In this work, we propose a team-based scheduling model to efficiently exploit parallelism between different or-parallel engines running on top of clusters of multicores. Our proposal defines a layered approach where a second-level scheduler specifies a clean interface for scheduling work between the base or-parallel engines, thus enabling different scheduling combinations to be used for distributing work among workers inside a team and among teams.

Keywords: Prolog, or-parallelism, environment copying, scheduling.

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*. The advantage of implicit parallelism is that one can develop specialized run-time systems to transparently explore the available parallelism in programs, thus freeing the programmers from the cumbersome task of explicitly identifying it.

Prolog programs offer 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. In both cases, we expect the parallel system to automatically identify opportunities for transforming parts of the computation (i.e., subgoals in a clause or clauses from a predicate) into concurrent instances of parallel work. These concurrent instances can be seen as independent Prolog engines which cooperate in the

parallel execution of the main program. We will often refer to these Prolog engines as *workers*.

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 and productive to exploit than and-parallelism, 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 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].

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 thus propose a novel computational model to efficiently exploit or-parallelism for clusters of low cost multicore architectures. Nowadays, 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 [5], 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 and 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.

To address the combination, one alternative would be to design a novel model from scratch trying to unite some of the techniques from the existent shared and distributed memory based models. Another alternative, is our proposal that introduces a layered approach with two levels of computational units, *single engines (or workers)* and *or-parallel engines (or teams of workers)*. At the worker level, or-parallelism is explored by using the already available models for scheduling/distributing work among workers inside a team. For single multicore architectures or clusters of single core/processor architectures, our layered approach should thus achieve similar performance since it simply reuses the currently available or-parallel models. To schedule/distribute work between teams, we intro-

duce a second-level team-based scheduler and we specify a clean interface for scheduling work between the or-parallel engines, thus enabling different scheduling combinations to be used for distributing work among workers inside a team and among teams. Our proposal resembles the concept of teams used by some models combining and-parallelism with or-parallelism, like the Andorra-I [13] or ACE [6] systems, where a layered approach also implements different schedulers to deal with each level of parallelism.

The remainder of the paper is organized as follows. First, we introduce some background about or-parallelism and the environment copying model. Next, we introduce our layered approach and discuss the major design issues, algorithms and challenges. Last, we advance 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 [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. Or-Parallelism

Or-parallelism arises when a subgoal call unifies with more than one of the clauses defining the predicate for the subgoal call at hand. In such a case, the parallel execution of the bodies of the alternative matching clauses corresponds to the exploitation of or-parallelism. A convenient way to visualize or-parallelism is through the *or-parallel search tree*. Figure 1 illustrates the or-parallel search tree for a small logic program and the query goal *a(X,Y)*.

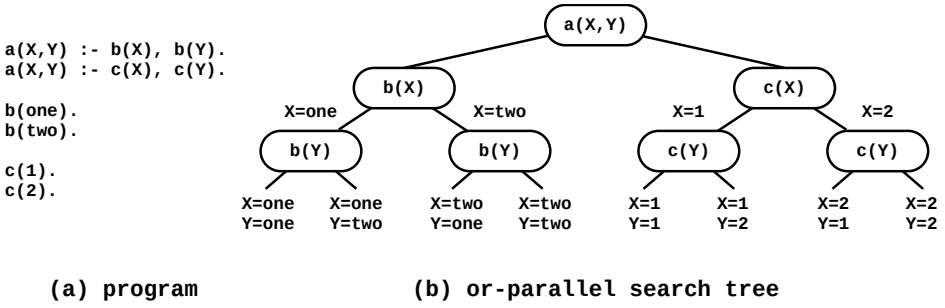

Fig. 1. An or-parallel search tree

The query goal *a(X,Y)* is non-deterministic since it unifies with both clauses of predicate *a/2*. In turn, the execution of each clause for *a/2* leads, respectively, to subgoal calls to predicates *b/1* and *c/1*, which are also non-deterministic. In Fig. 1, the search tree nodes represent the non-deterministic subgoal calls and the multiple alternatives in such nodes correspond to or-work that can be exploited in parallel. Or-parallelism is thus an efficient way of searching for alternative answers to a query and it frequently arises in applications that explore a large search space via backtracking. This is the typical case in applica-

tion areas such as expert systems, optimization and relaxation problems, certain types of parsing, natural language processing, deductive database systems, among others.

Intuitively, or-parallelism seems simple to implement since the various alternative branches of the or-tree are independent of each other. However, parallel execution can result in conflicting bindings for common variables. For any two different branches of the or-tree, there are a set of ancestor nodes than are common to both branches. A variable created in one of these common nodes might be bound differently in the two branches. In Fig. 1, we can see that the two variables X and Y, created in the top node, have different combinations of bindings in different branches. The environments of alternative branches have to be organized in such a way that conflicting bindings can be easily discernible. This problem, known as the *multiple bindings problem*, is a major problem when implementing or-parallelism.

3. Environment Copying

One of the most successful or-parallel models that solves the multiple bindings problem is the *environment copying* model. In the environment copying model, each worker keeps a separate copy of its own environment, thus the bindings to common 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.

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.

At the engine level, the search tree nodes are implemented as *choice points* in the local stack [1]. A choice point stores the open alternatives left to try. A choice point frame is pushed onto the local stack when a subgoal call unifies with more than one candidate clause and it is popped off when the last alternative clause is taken for execution. A choice point contains the necessary information to restore the state of the computation back to when the first clause was entered; plus a reference to the next open clause to try, in case the current one fails.

3.1. Incremental Copying

To reduce the overhead of stack copying when sharing work, 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 from the sharing worker 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 work. To implement incremental copying, Q should start by backtracking to the youngest common choice point 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 choice point 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 choice point 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 choice point [3].

3.2. 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]. Figure 2 shows a schematic representation of the sharing process between two workers using or-frames.

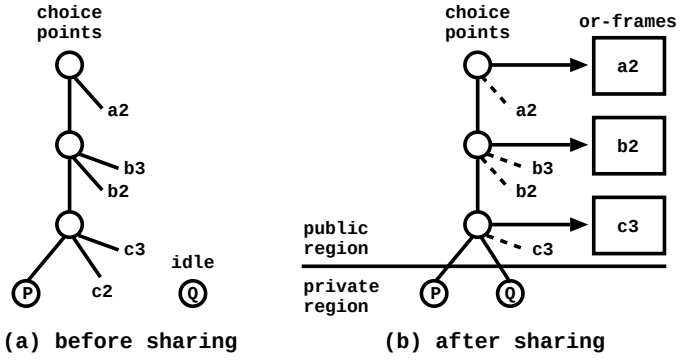

Fig. 2. Or-frames and public choice points

A worker sharing work adds an or-frame data structure to each private choice point made public. Each or-frame stores the reference 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 become represented by the newly created or-frames and by a *getwork* pseudo-alternative. Backtracking to a public choice point will thus always trigger the execution of the *getwork* pseudo-alternative and its execution allows for a synchronized access to the unexplored alternatives among the workers sharing the corresponding or-frame. The set of all or-frames form a tree that represents the public search or-tree.

3.3. 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. This matches the problem of having a pool of tasks to compute and either (i) access the pool to get the next available task every time a worker runs out of work (dynamic scheduling) or (ii) divide the tasks between the available by assigning each task to a specific worker (static scheduling).

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 3 illustrates the effect of these strategies in a work sharing operation between a busy worker P and an idle worker Q.

Figure 3(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 3(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 3(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 3(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 3(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.

3.4. The Yap Prolog System

The Yap Prolog system [12] 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 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

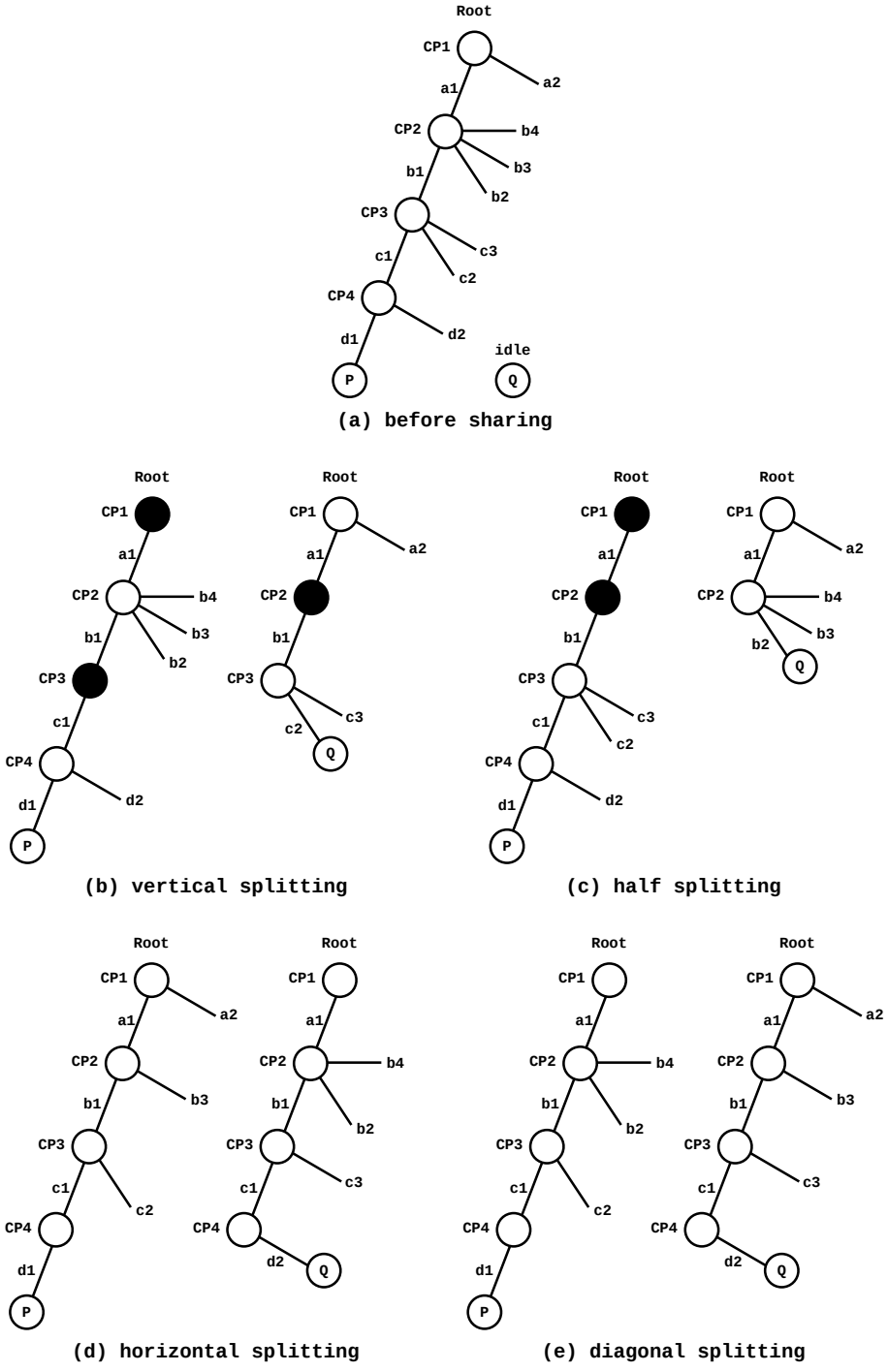

Fig. 3. Alternative stack splitting strategies

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. 4(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 youngest public 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. 4(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.

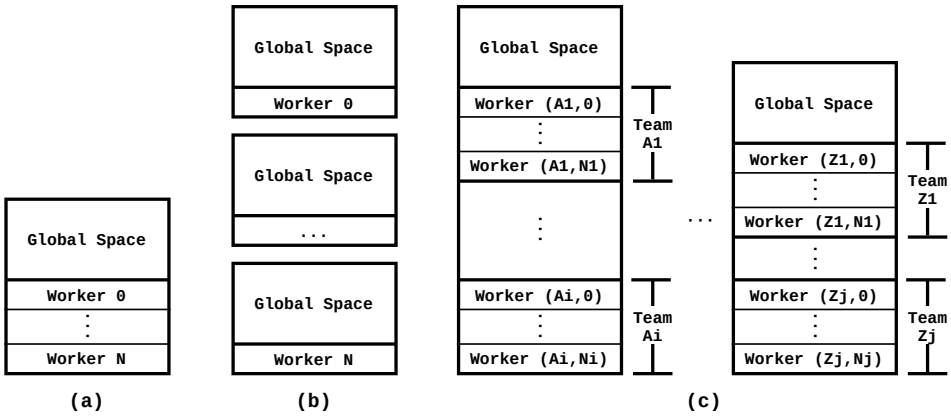

Fig. 4. Memory layout for: (a) workers in shared memory; (b) workers in distributed memory; and (c) teams of workers in clusters of multicores

4. Layered Approach

The goal behind our proposal is to implement the concept of teams and specify a clean interface for scheduling work between 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 im-

plies 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.

For workers inside a team, we can thus distribute work using both dynamic or static scheduling of work. For distributing work among teams, we can apply any of the four (static) stack splitting strategies described before. 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.

4.1. Memory Organization

In order to support teams, there are several changes that need to be made. An important one is the memory organization of the system. Figure 4(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. 4(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. 4(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.

4.2. 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 5 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 3 and 4 workers each. The computer node $N2$ has only one team, team C with 10 workers.

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

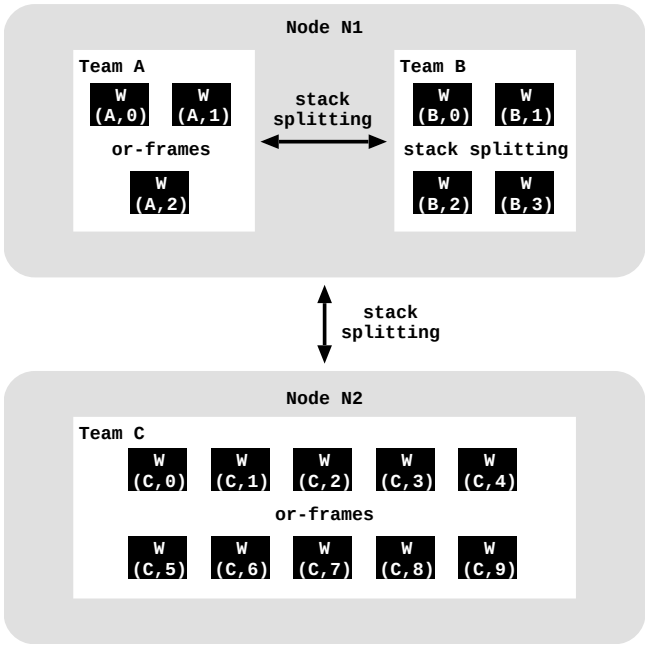

Fig. 5. Work scheduling within and among teams

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.

4.3. 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 6 shows a schematic representation of the sharing process between teams. Consider the cluster configuration in Fig. 5 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 6(a) shows the state of team A before the sharing request from C. The three workers in team A are executing in the private region of the search tree and all share the youngest three choice points. The top public choice point is already dead, i.e., without open alternatives, but the second and third public choice points have two ($b2$ and $b3$) and one ($c4$) open alternatives, respectively.

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 public) 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 public 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. 6(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 public 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 6(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.

4.4. Load Balancing

An important goal of our team-based scheduler is to achieve an efficient distribution of work between the different teams in order to optimize resource usage and thus minimize

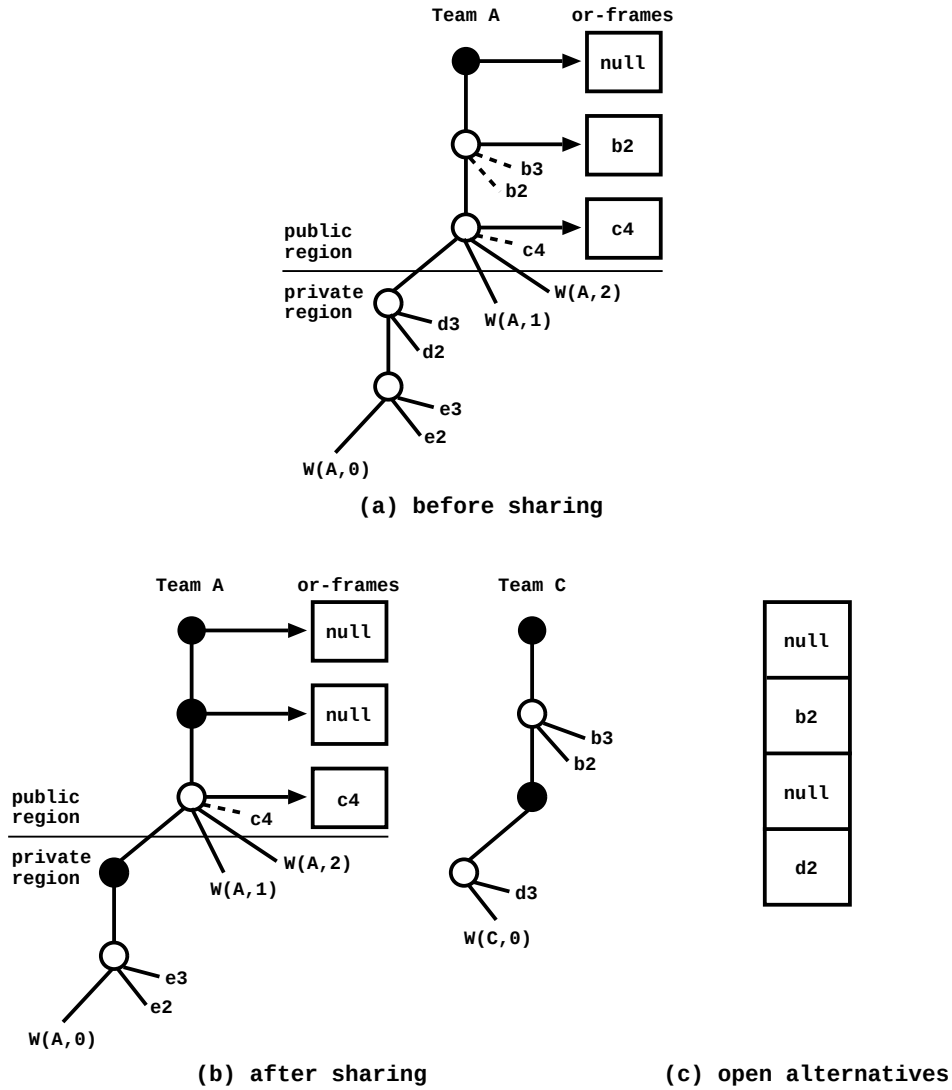

Fig. 6. Schematic representation of the sharing process between workers of different teams: in (a) we can see the configuration of team A when team C asks for work and in (b) we can see the configuration of both teams after the sharing process, considering that worker $W(A, 0)$ used vertical splitting to share its available work (in (c) we can see the array of open alternatives being shared) with worker $W(C, 0)$

response time. A good strategy, when searching for a team to request work, would be to select the busy team that holds the highest *work load* and that is *nearest to* the idle team. The work load is a measure of the amount of open alternatives and being near corresponds to a closer position in the search tree. Nevertheless, selecting such a team would require having precise information about the position and work load of all workers in all teams, which cannot be possible without introducing a considerable communication and synchronization overhead. A more reasonable solution is to find a compromise between the load balancing efficiency and the overhead required to support the implementation.

In our proposal, each worker holds a private *load register*, as a measure of the exact number of open alternatives in its current branch, and each team holds a private *load vector*, as a measure of the estimated work load of each fellow team. The load register keeps its functionality [10, 15, 14] and is used to support the process of selecting a busy worker to request work from, when a teammate runs out of work. The load vector is used to support the process of selecting a busy team to request work, when another team runs out of work.

The load vector is updated in two situations: when requesting work and when detecting termination. A trivial case occurs when receiving a sharing request, a zero work load can be automatically inferred for the requesting team. We do not introduce specific messages to explicitly ask for work load information and, instead, the existing messages are extended to include that information. When receiving a sharing request, the answering message is extended to include the work load of the answering team. When detecting termination, the termination tokens are extended to include the work load of all teams. The following section shows more details about the load vector usage.

5. Algorithms

In this section, we present in more detail the set of algorithms that implement the key aspects of our proposal.

5.1. Searching for Available Work

Algorithm 1 shows the pseudo-code for the $GetWorkFromWorker()$ 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 4). Next, it sends a share request to B (line 5) 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 6). After sharing, W returns to Prolog execution (line 7). Otherwise, if the sharing request gets refused, then W should try another busy worker from T, while there are teammates with available work (line 3).

On the other hand, if all workers in T run out of work (i.e., all workers in T are also executing the $GetWorkFromWorker()$ procedure), then one of the workers, named the *master worker*[1], will be selected to search for work from the other teams (line 8),

[1] Since all workers are out of work, there is no difference in selecting one or another worker to be the master worker and any worker can play that role.

Algorithm 1 $GetWorkFromWorker(W)$

```
1: T ← GetTeam(W)
2: while TeamNotFinished(T) do
3:     while HasAvailableWork(T) do
4:         B ← SelectBusyWorker(T)
5:         if SendWorkerRequest(B) = ACCEPT then
6:             InstallWorkFromWorker(W, B)
7:             return true
8:     if W = MasterWorker(T) then {W will search for work from the other teams}
9:         if GetWorkFromTeam(W) then {W has obtained work from another team}
10:            return true
11:        else {all teams should finish execution}
12:            SetTeamAsFinished(T)
13: return false
```

and for that it executes the $GetWorkFromTeam()$ procedure (line 9). If the call to $GetWorkFromTeam()$ succeeds, this means that the master worker has obtained a new piece of work from another team and, in such case, it returns to Prolog execution to start exploiting it (line 10). Otherwise, if the call to $GetWorkFromTeam()$ fails, this means that all teams are out of work and, in such case, team T is set as finished (line 12) and all workers in T then finish execution by returning failure (line 13).

Algorithm 2 shows the pseudo-code for the $GetWorkFromTeam()$ procedure that, given a master worker W, searches for a new piece of work from the other teams. Initially, W uses its load vector to select a busy team S from the set of available teams to potentially share work with (lines 1–3). Next, it sends a share request to the selected team S (line 4) and, if the request gets accepted, it performs the work sharing procedure with S (line 5) and returns successfully (line 6). Otherwise, if the sharing request gets refused, then W updates the load vector for team S (line 8) and tries another busy team, while there are teams with available work (line 2). On the other hand, if all teams run out of work (i.e., all master workers are also executing the $GetWorkFromTeam()$ procedure), then W returns failure (line 9).

Algorithm 2 $GetWorkFromTeam(W)$

```
1: load[ ] ← GetLoadVector()
2: while TeamsWithAvailableWork(load[ ]) do
3:     S ← SelectBusyTeam(load[ ])
4:     if SendTeamRequest(S) = ACCEPT then
5:         InstallWorkFromTeam(W, S)
6:         return true
7:     else {request rejected}
8:         load[S] ← GetLoadFromAnswerRequest(S)
9: return false
```

Finally, Algorithm 3 shows the pseudo-code for checking if there are teams with available work. If no busy team exists in the given load vector, then the termination detection

mechanism is invoked (lines 1–2). When detecting termination, the load vector is updated accordingly to include the work load of all teams. If no busy team still exists in the updated load vector, then all teams are definitively out of work and the procedure returns failure (lines 3–4). Otherwise, it succeeds (line 5).

Algorithm 3 $TeamsWithAvailableWork(load[\,])$

```
1: if NoBusyTeam(load[ ]) then
2:     load[ ] ← TerminationDetection()
3:     if NoBusyTeam(load[ ]) then
4:         return false
5: return true
```

5.2. Handling Requests

We next present in Algorithm 4 the pseudo-code for handling sharing requests. Please remember that we may have to deal with requests from workers in other teams or from workers in the same team. The main difference between both situations is that team requests (the former case) are first delegated to the team's busy worker B with the highest work load (lines 2–5). This delegation is done by the first worker that checks for the request (line 2). The selected busy worker B will then be the one responsible to answer the request (lines 6–13).

Algorithm 4 $CheckRequests(W)$

```
 1: T ← GetTeam(W)
 2: if HasTeamRequest(T) then
 3:     R ← GetRequestingTeam()
 4:     B ← SelectBusyWorker(T)
 5:     DelegateTeamRequestToWorker(R, B)
 6: if HasDelegatedRequest(W) then
 7:     R ← GetRequestingTeam()
 8:     if HasAvailableWork(W) then
 9:         AnswerRequest(R, ACCEPT)
10:         ShareWorkWithTeam(W, R)
11:     else
12:         load ← GetLoad()
13:         AnswerRequest(R, REJECT, load)
14: if HasWorkerRequest(W) then
15:     I ← GetRequestingWorker()
16:     if HasAvailableWork(W) then
17:         AnswerRequest(I, ACCEPT)
18:         ShareWorkWithWorker(W, I)
19:     else
20:         AnswerRequest(I, REJECT)
```

Both situations are then very similar. A given worker W with pending sharing requests may accept or refuse them accordingly to its current work load. If W has available work to share, i.e., if its work load is higher than a threshold value, it accepts the sharing request and starts the sharing process by calling the $ShareWorkWithTeam()$ or the $ShareWorkWithWorker()$ procedure (lines 9–10 and 17–18 respectively). Otherwise, it replies with a negative answer (lines 13 and 20). In the case of a delegated (team) request, W includes in the answering message its current work load.

5.3. Work Sharing

At last, we present the algorithms for work sharing. We start with the work sharing operation between workers in the same team. Algorithm 5 shows the pseudo-code to be executed by the sharing worker and Algorithm 6 shows the pseudo-code to be executed by the requesting worker.

Algorithm 5 $ShareWorkWithWorker(W, I)$

```
1: stacks ← GetStacks(W)
2: if SchedulingMode() = STATIC then {W's team is using static scheduling}
3:     SendStacks(I, stacks)
4:     strategy ← GetSplittingStrategy()
5:     ApplyStackSplitting(stacks, strategy)
6: else {[W's team is using dynamic scheduling}
7:     PublishPrivateWork(stacks)
8:     orframe ← GetYoungerOrFrame(stacks)
9:     AddWorkerToPublicRegion(W, orframe)
10:    AddWorkerToPublicRegion(I, orframe)
11:    SendStacks(I, stacks)
```

Algorithm 6 $InstallWorkFromWorker(W, B)$

```
1: stacks ← RecvStacks(B)
2: InstallStacks(W, stacks)
3: if SchedulingMode() = STATIC then
4:     strategy ← GetSplittingStrategy()
5:     ApplyStackSplitting(stacks, strategy)
```

Given a sharing worker W and an idle worker I, the $ShareWorkWithWorker()$ procedure proceeds as follows. If W's team is using static scheduling, W starts by sending its execution stacks to I (line 3) and then it applies stack splitting to its own stacks accordingly to the splitting strategy in use (lines 4–5). Otherwise, if using or-frames, W turns public its private choice points (line 7), adds both workers to the or-frames in the public region (lines 8–10) and sends its execution stacks to I (line 11).

On the other hand, in the $InstallWorkFromWorker()$ procedure, the given idle worker W starts by receiving the execution stacks from the sharing worker B (line 1),

install the stacks (line 2) and, if using static scheduling, applies stack splitting accordingly to the splitting strategy in use (lines 3–5).

Finally, we discuss the work sharing operation between workers in different teams. Algorithm 7 shows the pseudo-code to be executed by the sharing worker and Algorithm 8 shows the pseudo-code to be executed by the requesting worker.

Algorithm 7 $ShareWorkWithTeam(W, R)$

```
 1: stacks ← GetStacks(W)
 2: cp ← GetYoungerChoicePoint(stacks)
 3: i ← GetChoicePointDepth(cp)
 4: strategy ← GetSplittingStrategyForSharingWorkBetweenTeams()
 5: if SchedulingMode() = STATIC then {W's team is using static scheduling}
 6:    repeat {store open alternatives to share with R}
 7:       alts[i] ← GetOpenAlternativeAndApplyStackSplitting(cp, strategy)
 8:       cp = PreviousChoicePoint(cp, stacks)
 9:       i ← i − 1
10:    until i
11: else {W's team is using dynamic scheduling}
12:    repeat {store open alternatives to share with R}
13:       if IsPrivateChoicePoint(cp) then
14:          alts[i] ← GetOpenAlternativeAndApplyStackSplitting(cp, strategy)
15:       else {cp is public}
16:          LockOrFrame(cp)
17:          alts[i] ← GetOpenAlternativeAndApplyStackSplitting(cp, strategy)
18:          UnlockOrFrame(cp)
19:       cp = PreviousChoicePoint(cp, stacks)
20:       i ← i − 1
21:    until i
22: SendStacks(R, stacks)
23: SendOpenAlternatives(R, alts[ ])
24: load ← GetLoad()
25: BcastLoadInfo(W, load)
```

Given a sharing worker W and a requesting team R, the $ShareWorkWithTeam()$ procedure proceeds as follows. If W is using static scheduling inside its team, then W starts by applying stack splitting to its own stacks (lines 6–10) accordingly to the splitting strategy to be used for sharing work between teams (line 4) and, at the same time, initializes the array with the open alternatives to be shared with R (line 7). Note that for sharing work between teams, we can apply any of the four stack splitting strategies described before. Otherwise, if W's team is using or-frames, the procedure is similar. The main difference is that for public choice points we must synchronize the access to the corresponding or-frame (lines 16–18) before applying the stack splitting strategy to be used for sharing work between teams and initializing the array with the open alternatives to be shared with R (lines 12–21). In both cases, at the end, W sends the execution stacks and the array with the open alternatives to R and broadcasts its load info to all other teams (lines 22-25).

Algorithm 8 $InstallWorkFromTeam(W, S)$

```
1: stacks ← RecvStacks(S)
2: alts[ ] ← RecvOpenAlternatives(S)
3: InstallStacks(W, stacks)
4: cp ← GetYoungerChoicePoint(stacks)
5: i ← GetChoicePointDepth(cp)
6: repeat {install open alternatives}
7:    OpenAlternative(cp) ← alts[i]
8:    cp ← PreviousChoicePoint(cp, stacks)
9:    i ← i − 1
10: until i
```

In the $InstallWorkFromTeam()$ procedure, the given idle worker W starts by receiving the execution stacks and the array of open alternatives from the sharing worker in team S (lines 1–2), install the stacks (line 3) and install the open alternatives from the received array (lines 4–10).

6. Conclusions

We have proposed a novel computational model to efficiently exploit or-parallelism from the recent 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, our approach defines a layered approach where a team-based scheduler specifies a clean interface for scheduling work between the base or-parallel engines, thus enabling different scheduling combinations to be used for distributing work 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) support incremental copying between teams; (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. The authors want to thank thank the anonymous reviewers for their comments/suggestions and appreciate the guidance in making the paper clearer. 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 SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281). Joao Santos is funded by the FCT grant SFRH/BD/76307/2011.

References

1. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)

2. Ali, K., Karlsson, R.: Full Prolog and Scheduling OR-Parallelism in Muse. International Journal of Parallel Programming 19(6), 445–475 (1990)
3. Ali, K., Karlsson, R.: The Muse Approach to OR-Parallel Prolog. International Journal of Parallel Programming 19(2), 129–162 (1990)
4. Gupta, G., Pontelli, E.: Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In: International Conference on Logic Programming. pp. 290–304. The MIT Press (1999)
5. Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.V.: Parallel Execution of Prolog Programs: A Survey. ACM Transactions on Programming Languages and Systems 23(4), 472–602 (2001)
6. Gupta, G., Pontelli, E., Hermenegildo, M.V., Santos Costa, V.: ACE: And/Or-parallel Copying-based Execution of Logic Programs. In: International Conference on Logic Programming. pp. 93–109. The MIT Press (1994)
7. Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B.: The Aurora Or-Parallel Prolog System. In: International Conference on Fifth Generation Computer Systems. pp. 819–830. Institute for New Generation Computer Technology (1988)
8. Pontelli, E., Villaverde, K., Guo, H.F., Gupta, G.: 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. Rocha, R., Silva, F., Martins, R.: YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In: Portuguese Conference on Artificial Intelligence. pp. 136–150. No. 2902 in LNAI, Springer-Verlag (2003)
10. Rocha, R., Silva, F., Santos Costa, V.: YapOr: an Or-Parallel Prolog System Based on Environment Copying. In: Portuguese Conference on Artificial Intelligence. pp. 178–192. No. 1695 in LNAI, Springer-Verlag (1999)
11. Santos Costa, V., Dutra, I., Rocha, R.: 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. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming 12(1 & 2), 5–34 (2012)
13. Santos Costa, V., Warren, D.H.D., Yang, R.: 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. pp. 83–93. ACM (1991)
14. Vieira, R., Rocha, R., Silva, F.: On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. In: Colloquium on Implementation of Constraint and LOgic Programming Systems. pp. 71–85 (2012)
15. Vieira, R., Rocha, R., Silva, F.: 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. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In: International Conference on Logic Programming. pp. 27–42. No. 2237 in LNCS, Springer-Verlag (2001)
17. Villaverde, K., Pontelli, E., Guo, H., Gupta, G.: A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In: International Euro-Par Conference. pp. 694–703. No. 2790 in LNCS, Springer-Verlag (2003)

Joao Santos received his BSc in Computer Science in 2008 and his MSc in Network and Information Systems Engineering in 2010, both degrees from Faculty of Science of the

University of Porto. Since 2009, he has been a researcher at the CRACS & INESC TEC research unit, collaborating in projects STAMPA and LEAP. In 2012, he started his PhD in Computer Science at University of Porto and his main research topic is Parallelism in Logic Environments.

Ricardo Rocha is an Assistant Professor at the Department of Computer Science, Faculty of Sciences, University of Porto, Portugal and a researcher at the CRACS & INESC TEC research unit. He received his PhD degree in Computer Science from the University of Porto in 2001 and his main research topics are the Design and Implementation of Logic Programming Systems, Tabling in Logic Programming and Parallel and Distributed Computing. Another areas of interest include Inductive Logic Programming, Probabilistic Logic Programming and Deductive Databases. He is also one of the main developers of Yap Prolog system, and in particular of the execution models that support tabling and parallel evaluation. He has published more than 50 refereed papers in journals and international conferences, has supervised 11 MSc students and has leading role in two national projects: project STAMPA, funded with 150,000 Euros, and project LEAP, funded with 115,000 Euros. Currently, he also serves the ALP Newsletter as area co-editor for the Implementations and Systems track.

Received: October 25, 2013; Accepted: May 26, 2014.