High Level Thread-Based Competitive Or-Parallelism in Logtalk[⋆]

Paulo Moura[1,3], Ricardo Rocha[2,3], and Sara C. Madeira[1,4]

[1] Dep. of Computer Science, University of Beira Interior, Portugal
`{pmoura,smadeira}@di.ubi.pt`
[2] Dep. of Computer Science, University of Porto, Portugal
`ricroc@dcc.fc.up.pt`
[3] Center for Research in Advanced Computing Systems, INESC–Porto, Portugal
[4] Knowledge Discovery and Bioinformatics Group, INESC–ID, Portugal

Abstract. This paper presents the logic programming concept of *thread-based competitive or-parallelism*, which combines the original idea of competitive or-parallelism with committed-choice nondeterminism and speculative threading. In thread-based competitive or-parallelism, an explicit disjunction of subgoals is interpreted as a set of concurrent alternatives, each running in its own thread. The individual subgoals usually correspond to predicates implementing different procedures that, depending on the problem specifics, are expected to either fail or succeed with different performance levels. The subgoals compete for providing an answer and the first successful subgoal leads to the termination of the remaining ones. We discuss the implementation of thread-based competitive or-parallelism in the context of Logtalk, an object-oriented logic programming language, and present experimental results.

Keywords: Or-parallelism, speculative threading, implementation.

1 Introduction

Or-parallelism is a simple form of parallelism in logic programs [1], where the bodies of alternative clauses for the same goal are executed concurrently. Or-parallelism is often explored *implicitly*, possibly with hints from the programmer to guide the system. Common uses include search-oriented applications, such as parsing, database querying, and data mining. In this paper, we introduce a different, explicit form of or-parallelism, *thread-based competitive or-parallelism*, that combines the original idea of competitive or-parallelism [2] with committed-choice nondeterminism [3] and speculative threading [4]. Committed-choice nondeterminism, also known as *don't-care* nondeterminism, means that once an alternative is taken, the computation is committed to it and cannot backtrack or explore in parallel other alternatives. Committed-choice nondeterminism is useful whenever a single solution is sought among a set of potential alternatives.

[⋆] This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and MOGGY (PTDC/EIA/70830/2006).

Speculative threading allows the exploration of different alternatives, which can be interpreted as competing to provide an answer for the original problem. The key idea is that multiple threads can be started without knowing *a priori* which of them, if any, will perform useful work. In competitive or-parallelism, different alternatives are interpreted as competing for providing an answer. The first successful alternative leads to the termination of the remaining ones. From a declarative programming perspective, thread-based competitive or-parallelism allows the programmer to specify alternative procedures to solve a problem without caring about the details of speculative execution and thread handling. Another key point of thread-based competitive or-parallelism is its simplicity and implementation portability when compared with classical or-parallelism implementations. The ISO Prolog multi-threading standardization proposal [5] is currently implemented in several systems including SWI-Prolog, Yap and XSB, providing a highly portable solution given the number of operating systems supported by these Prolog systems. In contrast, most or-parallelism systems described in the literature [1] are no longer available, due to the complexity of maintaining and porting their implementations.

Our research is driven by the increasing availability of multi-core computing systems. These systems are turning into a viable high-performance, affordable and standardized alternative to the traditional (and often expensive) parallel architectures. The number of cores per processor is expected to continue to increase, further expanding the areas of application of competitive or-parallelism.

The remainder of the paper is organized as follows. First, we present in more detail the concept of competitive or-parallelism. Second, we discuss the implementation of competitive or-parallelism in the context of Logtalk [6], an object-oriented logic programming language, and compare it with classical or-parallelism. Next we present experimental results. Follows a discussion on how tabling can be used to take advantage of partial results gathered by speculative computations. We then identify further potential application areas where competitive or-parallelism can be useful. Finally, we outline some conclusions and describe further work. In the remaining of the paper, the expression *competitive or-parallelism* will be used interchangeably with the expression *thread-based competitive or-parallelism*.

2 Thread-Based Competitive Or-Parallelism

The concept of thread-based competitive or-parallelism is based on the interpretation of an *explicit disjunction of subgoals* as a set of concurrent alternatives, each running in its own thread. Each individual alternative is assumed to implement a different procedure that, depending on the problem specifics, is expected to either fail or succeed with different performance results. For example, one alternative may converge quickly to a solution, other may get trapped into a local, suboptimal solution, while a third may simply diverge. The subgoals are interpreted as competing for providing an answer and the first subgoal to complete leads to the termination of the threads running the remaining subgoals. The semantics of a competitive or-parallelism call are simple. Given a disjunction of subgoals, a competitive or-parallelism call blocks until one of the following situations occurs: one of the subgoals succeeds; all the subgoals fail; or one of the subgoals generates an exception. All the remaining threads are terminated once one of the subgoals succeeds or an exception is thrown during the execution of one of the running threads. The competitive or-parallelism call succeeds if and only if one of the subgoals succeeds. When one of the subgoals generates an exception, the competitive or-parallelism call terminates with the same exception.[1] When two or more subgoals generate exceptions, the competitive or-parallelism call terminates with one of the generated exceptions.

For example, assume that we have implemented several methods for calculating the roots of real functions.[2] In Logtalk, we may then write:

```
find_root(F, A, B, Error, Zero, Method) :-
    threaded((
        bisection::find_root(F, A, B, Error, Zero), Method = bisection
    ;   newton::find_root(F, A, B, Error, Zero), Method = newton
    ;   muller::find_root(F, A, B, Error, Zero), Method = muller
    )).
```

In this example, the competitive or-parallelism call (implemented by the Logtalk built-in meta-predicate `threaded/1`) returns both the identifier of the fastest successful method and its result. Depending on the function and on the initial interval, one method may converge quickly to the root of the function while the others may simply diverge. Thus, by avoiding committing to a specific method that might fail for some functions, the competitive or-parallelism call allows a solution to be found corresponding to the fastest, successful method.

Consider now a different example, the *generalized water jugs* problem. In this problem, we have several jugs of different capacities and we want to measure a certain amount of water. We may *fill* a jug, *empty* it, or *transfer* its contents to another jug. As in our previous example, we may apply several methods to solve this problem. The water jugs state-space can be explored using e.g. breadth-first, depth-first, or hill-climbing search strategies. We could write:

```
solve(WaterJug, Liters, Jug1, Jug2, Steps) :-
    threaded((
        depth_first::solve(WaterJug, Liters, Jug1, Jug2, Steps)
    ;   hill_climbing::solve(WaterJug, Liters, Jug1, Jug2, Steps)
    ;   breadth_first::solve(WaterJug, Liters, Jug1, Jug2, Steps)
    )).
```

Different heuristics could also be explored in parallel. As before, without knowing *a priori* the amount of water to be measured, we have no way of telling which method or heuristic will be faster. This example is used later in this paper to provide experimental results.

[1] If we want the computation to proceed despite the exception generated, we can convert exceptions into failures by wrapping the thread subgoal in a `catch/3` call.

[2] The full source code of this example is included in the current Logtalk distribution.

These examples illustrate how thread-based competitive or-parallelism distinguish itself from existing or-parallel systems by allowing fine-grained control at the goal level using an explicit parallelism construct. As in most implementations of or-parallelism, the effectiveness of competitive or-parallelism relies on several factors. These factors are discussed in detail next.

3 Implementation

In this section we discuss the implementation of competitive or-parallelism in the context of Logtalk, given the core predicates found on the ISO standardization proposal for Prolog threads [5].

3.1 Logtalk Support

Logtalk is an open source object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. Logtalk takes advantage of modern multi-processor and multi-core computers to support high level multi-threading programming, allowing objects to support both synchronous and asynchronous messages without bothering with the details of creating and destroying threads, implement thread communication, or synchronizing threads.

Competitive or-parallelism is implemented in Logtalk using the built-in meta-predicate `threaded/1`, which supports both competitive or-parallelism and independent (and quasi-independent) and-parallelism.

The `threaded/1` predicate proves a conjunction or disjunction of subgoals running each subgoal in its own thread.[3] When the argument is a conjunction of goals, a call to this predicate implements independent and-parallelism semantics. When the argument is a disjunction of subgoals, a call to this predicate implements the semantics of competitive or-parallelism, as detailed in the previous section. The `threaded/1` predicate is deterministic and opaque to cuts and, thus, there is no backtracking over completed calls.

The choice of using Prolog core multi-threading predicates to implement competitive or-parallelism provides several advantages in terms of simplicity and portability when compared with traditional, low-level or-parallelism implementation solutions. Nevertheless, three problems must be addressed when exploiting or-parallelism: (i) multiple binding representation, (ii) work scheduling, and (iii) predicate side-effects. These problems are addressed in the sections below.

3.2 Multiple Binding Representation

The multiple binding representation is a crucial problem for the efficiency of classical or-parallel systems. The concurrent execution of alternative branches

[3] The predicate argument is not flattened; parallelism is only applied to the outermost conjunction or disjunction. When the predicate argument is neither a conjunction nor a disjunction of subgoals, no threads are used. In this case, the predicate call is equivalent to a call to the ISO Prolog standard predicate `once/1`.

of the search tree can result in several conflicting bindings for shared variables. The main problem is that of efficiently representing and accessing *conditional bindings*.[4] The environments of alternative branches have to be organized in such a way that conflicting conditional bindings can be easily discernible.

The multiple binding representation problem can be 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 (see e.g [1]). Arguably, the two most successful ones are *environment copying*, as implemented in the Muse [7] and YapOr [8] systems, and *binding arrays*, as implemented in the Aurora system [9]. In the environment copying model, each worker maintains its own copy of the environment (stack, heap, trail, etc) in which it can write without causing binding conflicts. In this model, even unconditional bindings are not shared. In the binding arrays model, each worker maintains a private array data structure, called the *binding array*, where it stores its conditional bindings. Each variable along a branch is assigned to a unique number that identifies its offset entry in the binding array.

In a competitive or-parallelism call, only the first successful subgoal in the disjunction of subgoals can lead to the instantiation of variables in the original call. This simplifies our implementation as the Prolog core support for multi-threading programming can be used straightforward. In particular, we can take advantage of the Prolog thread creation predicate `thread_create/3`. Each new Prolog thread created by this predicate runs a *copy* of the goal argument using its own set of data areas (stack, heap, trail, etc). Its implementation is similar to the environment copying approach but simpler as only the goal is copied. As each thread runs a copy of the goal, no variables are shared across threads. Thus, the bindings of shared variables occurring within a thread are independent of bindings occurring in other threads. This operational semantics simplifies the problem of multiple binding representation in competitive or-parallelism, which results in a simple implementation with only a small number of lines of Prolog source code. Nevertheless, because each thread is running a copy of the original goal, thus breaking variable bindings, we need a solution for retrieving the bindings of the successful thread; our implementation solution is presented later. Copying a goal into a thread and copying the successful bindings back to the following computations may result in significant overhead for goals with large data structures arguments. Thus, we sacrifice some performance in order to provide users with an high-level, portable implementation.

3.3 Work Scheduling

Even though the cost of managing multiple environments cannot be completely avoided, it may be minimized if the *operating-system's scheduler* is able to divide efficiently the available work between the available computational units during execution. In classical or-parallelism, the or-parallel system usually knows the

[4] A binding of a variable is said to be *conditional* if the variable was created before the last choice point, otherwise it is said to be *unconditional*.

number of computational units (processors or cores) that are available in the supporting architecture. A high-level scheduler then uses this number to create an equal number of workers (processes or threads) to process work. The scheduler's task of load balancing and work dispatching, from the user's point-of-view, is completely *implicit*, i.e., the user cannot interfere in the way work is scheduled for execution. This is a nice property as load balancing and work dispatching are usually complex tasks due to the dynamic nature of work.[5]

In competitive or-parallelism, the problem of work scheduling differs from classical or-parallelism due to the use of explicit parallelism. The system can also know the number of computational units that are available in the supporting architecture, but the user has *explicit* control over the process of work dispatching. This explicit control can lead to more complex load balancing problems, as the number of running workers (threads) can easily exceed the number of available computational units (processors or cores). Our current implementation delegates load balancing to the operating-system thread scheduler. However, we can explicitly control the number of running threads using parametric objects with a parameter for the maximum number of running threads. This is a simple programming solution, used in most of the Logtalk multi-threading examples.

In classical or-parallelism, another major problem for scheduling is the presence of pruning operators like the cut predicate. When a cut predicate is executed, all alternatives to the right of the cut are pruned, therefore never being executed in a sequential system. However, in a parallel system, the work corresponding to these alternatives can be picked for parallel execution before the cut is executed, therefore resulting in wasted computational effort when pruning takes place. This form of work is known as *speculative work* [14]. An advanced scheduler must be able to reduce to a minimum the speculative computations and at the same time maintain the granularity of the work scheduled for execution [15,16].

In competitive or-parallelism, the concept of speculative work is part of its operational semantics, not because of the cut's semantics as the `threaded/1` predicate is deterministic and opaque to cuts, but because of the way subgoals in a competitive or-parallelism call are terminated once one of the subgoals succeeds. In this case, the speculative work results from the computational effort done by the unsuccessful or slower threads when pruning takes place. We can view the `threaded/1` predicate as an high-level *green cut* predicate that prunes all the alternatives to the left and to the right of the successful subgoal. For now, we have postponed working on an advanced, high-level scheduler.

3.4 Thread Results and Cancellation Issues

Logtalk multi-threading support uses message queues to collect thread results. This allows execution to be suspended while waiting for a thread result to be posted to a message queue, avoiding polling, which would hurt performance.

[5] A number of different scheduling strategies have been proposed to efficiently deal with this problem on classical or-parallelism; see e.g. [10,11,12,13].

Each thread posts its result to the message queue of the parent thread within the context of which the competitive or-parallelism call is taking place. The results posted by each thread are tagged with the identifiers of the remaining threads of the competitive or-parallelism call. This allows the cancellation of the remaining threads once a successful result (or an exception) is posted. In Logtalk, a template with the thread tags and the original disjunction subgoals is constructed when compiling a competitive or-parallelism call. The template thread tags are instantiated at run-time using the identifiers of the threads created when executing the competitive or-parallelism call. The first successful thread unifies its result with the corresponding disjunction goal in the template, thus retrieving any variable bindings resulting from proving the competitive or-parallelism call.[6]

The effectiveness of competitive or-parallelism relies on the ability to cancel the slower threads once a winning thread completes (or throws an exception), as they would no longer be performing useful work. But canceling a thread may not be possible and, when possible, may not be as fast as desired if a thread is in a state where no interrupts are accepted. In the worst case scenario, some slower threads may run up to completion. Canceling a thread is tricky in most low-level multi-threading APIs, including POSIX threads. Thread cancellation usually implies clean-up operations, such as deallocating memory, releasing mutexes, flushing and possibly closing of opened streams.

In Prolog, thread cancellation must occur only at safe points of the underlying virtual machine. In the case of the ISO Prolog multi-threading standardization proposal, the specified safe points are blocking operations such as reading a term from a stream, waiting for a message to be posted to a message queue, or thread sleeping. These blocking operations allow interrupt vectors to be checked and signals, such as thread cancellation, to be processed. Therefore, the frequency of blocking operations determines how fast a thread can be canceled. Fortunately, to these minimal set of cancellation safe points, the compilers currently implementing the proposal often add a few more, e.g., whenever a predicate enters its *call port* in the traditional box model of Prolog execution. In practical terms this means that, although tricky in its low-level implementation details, it is possible to cancel a thread whenever necessary. The standardization proposal specifies a predicate, `thread_signal/2`, that allows signaling a thread to execute a goal as an interrupt. Logtalk uses this predicate for thread cancellation. Some current implementations of this predicate fail to protect the processing of a signal from interruption by other signals. Without a solution for suspending further signals while processing an interrupt, there is the danger in corner cases of leaving dangling, zombie threads when canceling a thread whose goal recursively creates other threads. This problem is expected to be solved in the short term.

3.5 Side-Effects and Dynamic Predicates

The subgoals in a competitive or-parallelism call may have side-effects that may clash if not accounted for. Two common examples are input/output operations

[6] For actual implementation details and programming examples, the reader is invited to consult the sources of the Logtalk compiler, which are freely available online [17].

and asserting and retracting clauses for dynamic predicates. To prevent conflicts, Logtalk and the Prolog compilers implementing the ISO Prolog multi-threading standardization proposal allow predicates to be declared synchronized, thread shared (the default), or thread local. Synchronized predicates are internally protected by a mutex, thus allowing for easy thread synchronization. Thread private dynamic predicates may be used to implement thread local dynamic state.

In Logtalk, predicates with side-effects can be declared as synchronized by using the `synchronized/1` directive. Calls to synchronized predicates are protected by a mutex, thus allowing for easy thread synchronization. For example:

```
:- synchronized(db_update/1).    % ensure thread synchronization
db_update(Update) :- ...         % predicate with side-effects
```

A dynamic predicate, however, cannot be declared as synchronized. In order to ensure atomic updates of a dynamic predicate, we need to declare as synchronized the predicate performing the update.

The standardization proposal specifies that, by default, dynamic predicates are shared by all threads. Thus, any thread may call and may assert and retract clauses for the dynamic predicate. The Prolog compilers that implement the standardization proposal allow dynamic predicates to be instead declared thread local.[7] Thread-local dynamic predicates are intended for maintaining thread-specific state or intermediate results of a computation. A thread local predicate directive tells the system that the predicate may be modified using the built-in assert and retract predicates during execution of the program but, unlike normal shared dynamic data, each thread has its own clause list for the predicate (this clause list is empty when a thread starts). Any existing predicate clauses are automatically reclaimed by the system when the thread terminates.

4 Experimental Results

We chose the *generalized water jug* problem to provide the reader with some experimental results for competitive or-parallelism. In this problem, two water jugs with p and q capacities are used to measure a certain amount of water. A third jug is used as an accumulator. When p and q are relatively prime, it is possible to measure any amount of water between 1 and $p + q$ [18]. This is a classical state-space search problem, which we can try to solve using blind or heuristic search methods. In this experiment, we used competitive or-parallelism (COP) to simultaneously explore depth-first (DF), breadth-first (BF), and hill-climbing (HC) search strategies. Depending on the values of p and q, the required number of steps to measure a given amount of water can range from two steps (in trivial cases) to several dozens of steps.[8] Moreover, the number of potential nodes to explore can range from a few nodes to hundreds of thousands of nodes.

[7] Due to syntactic differences between these Prolog compilers, directives for specifying both thread local and thread shared dynamic predicates are not yet specified in the standardization proposal.

[8] There is an upper bound to the number of steps necessary for measuring a certain amount of water [19]. In this simple experiment we ignored this upper bound.

Our experimental setup used Logtalk 2.33.0 with SWI-Prolog 5.6.59 64 bits as the back-end compiler on an Intel-based computer with four cores and 8 GB of RAM running Fedora Core 8 64 bits.[9] Table 1 shows the running times, in seconds, when 5-liter and 9-liter jugs were used to measure from 1 to 14 liters of water. It allows us to compare the running times of single-threaded DF, HC, and BF search strategies with the COP multi-threaded call where one thread is used for each individual search strategy. The results show the average of thirty runs. We highlight the fastest method for each measure. The last column shows the number of steps of the solution found by the competitive or-parallelism call. The maximum solution length was set to 14 steps for all strategies. The time taken to solve the problem ranges from 0.000907 to 8.324970 seconds. Hill climbing is the fastest search method in six of the experiments. Breadth-first comes next as the fastest search method in five experiments. Depth-first search is the fastest search method only in three experiments. Repeating these experiments with other capacities for the water jugs yields similar results.

Table 1. Measuring from 1 to 14 liters with 5-liter and 9-liter jugs

Liters	**DF**	**HC**	**BF**	**COP**	**Overhead**	**Steps**
1	26.373951	0.020089	**0.007044**	0.011005	0.003961	5
2	26.596118	12.907172	**8.036822**	8.324970	0.288148	11
3	20.522287	**0.000788**	1.412355	0.009158	0.008370	9
4	20.081001	**0.000241**	0.001437	0.002624	0.002383	3
5	**0.000040**	0.000240	0.000484	0.000907	0.000867	2
6	3.020864	0.216004	**0.064097**	0.098883	0.034786	7
7	3.048878	**0.001188**	68.249278	0.008507	0.007319	13
8	2.176739	**0.000598**	0.127328	0.007720	0.007122	7
9	2.096855	**0.000142**	0.000255	0.003799	0.003657	2
10	**0.000067**	0.009916	0.004774	0.001326	0.001295	4
11	**0.346695**	5.139203	0.587316	0.404988	0.058293	9
12	14.647219	**0.002118**	10.987607	0.010785	0.008667	14
13	0.880068	0.019464	**0.014308**	0.029652	0.015344	5
14	0.240348	0.003415	**0.002391**	0.010367	0.007976	4

These results show that the use of competitive or-parallelism allows us to quickly find a sequence of steps of acceptable length to solve different configurations of the water jug problem. Moreover, given that we do not know *a priori* which search method will be the fastest for a specific measuring problem, competitive or-parallelism is a better solution than any of the individual search methods.

The overhead of the competitive or-parallelism calls is due to the implicit thread and memory management, plus low-level Prolog synchronization tasks.

The asynchronous nature of thread cancellation implies a delay between the successful termination of a thread and the cancellation of the other competing threads. Moreover, the current Logtalk implementation only returns the result

[9] The experiments can be easily reproduced by the reader by running the query `logtalk_load(mtbatch(loader)), mtbatch(swi)::run(search, 30).`

of a competitive or-parallelism call after all spawned threads are terminated. An alternative implementation where cancelled threads are detached in order to avoid waiting for them to terminate and being joined proved tricky and unreliable due to the reuse of thread identifiers by the back-end Prolog compilers.

The initial thread data area sizes and the amount of memory that must be reclaimed when a thread terminates can play a significant role on observed overheads, depending on the Prolog compiler memory model and on the host operating system. Memory allocation and release is a possible contention point at the operating-system level, as testified by past and current work on optimized, multi-threading aware memory allocators. (see e.g. [20]).

Low-level SWI-Prolog synchronization tasks also contribute to the observed overheads. In the current SWI-Prolog version, dynamic predicates are mutex locked even when they are declared thread local (in this case collisions occur when accessing the shared data structures used by SWI-Prolog to find and update local predicate definitions). Logtalk uses dynamic predicates to represent the method lookup caches associated with dynamic binding. While in previous versions the lookup caches are thread shared, the current Logtalk release uses thread local lookup caches. This change had a small impact on performance in Linux but provided a noticeable performance boost on MacOS X. Table 2 shows the results for the dynamic predicate used for the main lookup cache when running the query `mtbatch(swi)::run(search, 20)` in both Linux and MacOS.

Table 2. Mutex locks and collisions

	Linux		**MacOS X**	
	Locks	Collisions	Locks	Collisions
Thread shared	1022796427	3470000	907725567	17045455
Thread local	512935818	846213	458574690	814574

Thus, by simply making the lookup caches thread local, we reduced the number of collisions by 75% in Linux and 94% in MacOS X. Although different hardware is used in each case, is worth noting that, with a thread shared lookup cache, the number of collisions in MacOS X is five times the number of collisions in Linux. This is in accordance with our experience with other multi-threading tests where the Linux implementation of POSIX threads consistently outperforms that of MacOS X (and also the emulation of POSIX threads in Windows).

We are optimizing our implementation in order to minimize the thread management overhead. There is also room for further optimizations on the Prolog implementations of core multi-threading support. Nevertheless, even with the current implementations, our experimental results are promising.

5 The Role of Tabling

In complex problems, such as the ones discussed in the previous section, some of the competing threads, even if not successful, may generate intermediate results useful to other threads. Thus, dynamic programming in general, and tabling [21,22] in particular, is expected to play an important role in effective problem solving when using competitive or-parallelism.

In multi-threading Prolog systems supporting tabling, tables may be either private or shared between threads. In the latter case, a table may be shared once completed or two or more threads may collaborate in filling it. For applications using competitive or-parallelism, the most interesting uses of tabling will likely require the use of shared tables.

While thread-private tables are relatively easy to implement, all other cases imply sharing a dynamic data structure between threads, with all the associated issues of locking, synchronization, and potential deadlock cases. Thus, despite the availability of both threads and tabling in Prolog compilers such as XSB, Yap, and recently Ciao [23], the implementation of these two features such that they work together seamlessly implies complex ties to one another and to the underlying Prolog virtual machine. Nevertheless, promising results are described in a recent PhD thesis [24] and currently implemented in XSB [25]. In the current Yap version, tabling and threads are incompatible features; users must chose one or the other when building Yap. Work is planned to make Yap threads and tabling compatible. Ciao features a higher-level implementation of tabling when compared with XSB and Yap, which requires minimal modifications to the compiler and the abstract machine. This tabling support, however, is not yet available in the current Ciao stable release [26]. It will be interesting to see if this higher-level implementation makes the use of tabled predicates and thread-shared tables easier to implement in a multi-threading environment. These Prolog implementations are expected to eventually provide robust integration of these two key features. Together with the expected increase on the number of cores per processor, we can foresee a programming environment that provides all the building blocks for taking full advantage of competitive or-parallelism.

6 Potential Application Areas

Competitive or-parallelism support is useful when we have several algorithms to perform some computation and we do not know *a priori* which algorithm will be successful or will provide the best performance. This pattern is common to several classes of problems in different application areas. Problems where optimal solutions are sought may also be targeted when optimality conditions or quality thresholds can be incorporated in the individual algorithms.[10]

Interesting applications usually involve solving problems whose computational complexity implies using heuristic approaches with suboptimal solutions. In these cases, each thread in a competitive or-parallelism call can tackle a different starting point, or apply a different heuristic, in order to find a solution that, even if not optimal, is considered good enough for practical purposes.

A good example are biclustering applications (see e.g [27,28]), which provide multiple opportunities for applying speculative threading approaches, such as

[10] The cost or quality of the solutions being constructed by each algorithm may also be shared between threads.

competitive or-parallelism. Most instances of the biclustering problem are NP-hard. As such, most algorithmic approaches presented to date are heuristic and thus not guaranteed to find optimal solutions [27]. Common application areas include biological data analysis, information retrieval and text mining, collaborative filtering, recommendation systems, target marketing and database research. Given the complexity of the biclustering problem, the most promising algorithmic approaches are *greedy iterative search* and *distribution parameter identification* [27]. Both are amenable to speculative threading formulations.

Greedy iterative search methods are characterized by aggressively looking for a solution by making locally optimal choices, hoping to quickly converge to a globally good solution [29]. These methods may make wrong decisions and miss good biclusters when trapped in suboptimal solutions. Finding a solution satisfying a quality threshold often implies several runs using different starting points and possibly different greedy search strategies. Therefore, we may speculatively try the same or several greedy algorithms, with the same or different starting points, hopefully leading to different solutions satisfying a given quality threshold. In this case, the returned solution will be the first solution found that satisfies the quality metric and not necessarily the best solution. Note that this is a characteristic of greedy search, irrespective of the use of speculative threading.

Distribution parameter identification approaches assume that biclusters were generated from an underlying statistical model. Models are assumed to be defined by a fixed statistical distribution, whose set of parameters may be inferred from data. Different learning algorithms can be used to identify the parameters more likely to have generated the data [30]. This may be accomplished by iteratively minimizing a certain criterion. In this context, we can speculatively try different algorithms to infer the statistical model given the same initialization parameters, try different initialization parameters for the same distribution (using the same or different algorithms), or even assume different statistical models.

With the increasing availability of powerful multi-core systems, the parallel use of both greedy search and distribution parameter identification in biclustering applications is a promising alternative to current biclustering algorithms. In this context, high-level concepts, such as competitive or-parallelism, can play an important role in making speculative threading applications common place.

7 Conclusions and Future Work

We presented the logic programming concept of thread-based competitive or-parallelism, resulting from combining key ideas of competitive or-parallelism, committed-choice nondeterminism, speculative threading, and declarative programming. This concept is fully supported by an implementation in Logtalk, an open-source object-oriented logic programing language. We provided a description of our implementation and discussed its semantic properties, complemented with a discussion on thread cancellation and computational performance issues. This concept is orthogonal to the object-oriented features of Logtalk and can be implemented in plain Prolog and in non-declarative programming languages supporting the necessary threading primitives.

Competitive and classical or-parallelism target different classes of problems. Both forms of or-parallelism can be useful in non-trivial problems and can be supported in the same programming language. Competitive or-parallelism provides fine-grained, explicit control at the goal level of the tasks that should be executed in parallel, while classical parallel systems make use of implicit parallelism with possible parallelization hints at the predicate level.

For small problems, the benefits of competitive or-parallelism may not outweigh its inherent overhead. For computationally hard problems, this overhead is expected to be negligible. Interesting problems are characterized by the existence of several algorithms and heuristics, operating in a large search-space. In this context, we discussed potential applications where competitive or-parallelism can be a useful tool for problem solving.

Meaningful experimental results, following from the application of competitive or-parallelism to real-world problems, require hardware that is becoming common place. Consumer and server-level computers containing from two to sixteen cores, running mainstream operating-systems, are readily available. Each processor generation is expected to increase the number of cores, broadening the class of problems that can be handled using speculative threading in general, and competitive or-parallelism in particular.

Most research on speculative threading focus on low-level support, such as processor architectures and compiler support for automatic parallelization. In contrast, competitive or-parallelism is a high-level concept, targeted to programmers of high-level languages. In the case of Logtalk, thread-based competitive or-parallelism is supported by a single and simple to use built-in meta-predicate.

We found that core Prolog support for multi-threading programming provides all the necessary support for implementing Logtalk parallelism features. From a pragmatic perspective, this is an important result as (i) it leads to a simple, high-level implementation of both competitive or-parallelism and independent and-parallelism [31] that translates to only a few hundred lines of Prolog source code; (ii) it ensures wide portability of our implementation (the programmer can choose between SWI-Prolog, XSB, or Yap as the back-end compiler for exploring Logtalk multi-threading features on POSIX and Windows operating-systems).

Ongoing work focuses on improving and expanding the Logtalk support for multi-threading programming. In particular, we are fine-tuning the Logtalk implementation and working closely with the Prolog developers in the specification and implementation of the ISO standardization proposal for Prolog multi-threading programming. Major goals are minimizing the implicit overhead of thread management and testing our implementation for robustness in corner cases such as exhaustion of virtual memory address space in 32 bits systems. Future work will include exploring the role of tabling in competitive or-parallelism, implementing a load-balancing mechanism, and applying competitive or-parallelism to field problems.

Authors' Contributions. PM is the developer of the Logtalk language; he implemented the thread-based competitive or-parallelism concept, wrote the original draft of this paper and worked on the experimental results. RR contributed with

its knowledge on classical or-parallelism and tabling, helped with the comparison between the two concepts of or-parallelism, and made several contributions to the remaining sections of this paper. SM wrote the section on potential application areas; her current research on bioinformatics is one of the main motivations for this work. All authors read and approved the final manuscript.

Acknowledgments. We are grateful to Jan Wielemaker, Vítor Costa, Terrance Swift and Rui Marques for their groundwork implementing Prolog multi-threading core support and for helpful discussions on the subject of this paper.

A preliminary version of this work was published as a short paper in [32]. This paper provides additional implementation details, a detailed discussion of the experimental results, and additional sections on potential application areas and on the role of tabling on thread-based competitive or-parallelism. We thank the anonymous reviewers of our previous work for their helpful comments.

References

1. 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, 472–602 (2001)
2. Ertel, W.: Performance Analysis of Competitive Or-Parallel Theorem Proving. Technical report fki-162-91, Technische Universität München (1991)
3. Shapiro, E.: The Family of Concurrent Logic Programming Languages. ACM Computing Surveys 21, 413–510 (1989)
4. González, A.: Speculative Threading: Creating New Methods of Thread-Level Parallelization. Technology@Intel Magazine (2005)
5. Moura, P.: (ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Support), `http://logtalk.org/plstd/threads.pdf`
6. Moura, P.: Logtalk – Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior (2003)
7. Ali, K., Karlsson, R.: The Muse Approach to OR-Parallel Prolog. International Journal of Parallel Programming 19, 129–162 (1990)
8. Rocha, R., Silva, F., Santos Costa, V.: YapOr: an Or-Parallel Prolog System Based on Environment Copying. In: Barahona, P., Alferes, J.J. (eds.) EPIA 1999. LNCS (LNAI), vol. 1695, pp. 178–192. Springer, Heidelberg (1999)
9. 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, Institute for New Generation Computer Technology, pp. 819–830 (1988)
10. Calderwood, A., Szeredi, P.: Scheduling Or-parallelism in Aurora – the Manchester Scheduler. In: International Conference on Logic Programming, pp. 419–435. MIT Press, Cambridge (1989)
11. Ali, K., Karlsson, R.: Full Prolog and Scheduling OR-Parallelism in Muse. International Journal of Parallel Programming 19, 445–475 (1990)
12. Beaumont, A., Raman, S., Szeredi, P., Warren, D.H.D.: Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In: Aarts, E.H.L., van Leeuwen, J., Rem, M. (eds.) PARLE 1991. LNCS, vol. 506, pp. 403–420. Springer, Heidelberg (1991)

13. Sindaha, R.: Branch-Level Scheduling in Aurora: The Dharma Scheduler. In: International Logic Programming Symposium, pp. 403–419. MIT Press, Cambridge (1993)
14. Ciepielewski, A.: Scheduling in Or-parallel Prolog Systems: Survey and Open Problems. International Journal of Parallel Programming 20, 421–451 (1991)
15. Ali, K., Karlsson, R.: Scheduling Speculative Work in MUSE and Performance Results. International Journal of Parallel Programming 21, 449–476 (1992)
16. Beaumont, A., Warren, D.H.D.: Scheduling Speculative Work in Or-Parallel Prolog Systems. In: International Conference on Logic Programming, pp. 135–149. MIT Press, Cambridge (1993)
17. Moura, P.: (Logtalk), `http://logtalk.org`
18. Pfaff, T.J., Tran, M.M.: The generalized jug problem. Journal of Recreational Mathematics 31, 100–103 (2003)
19. Boldi, P., Santini, M., Vigna, S.: Measuring with jugs. Theoretical Computer Science 282, 259–270 (2002)
20. Berger, E.D., Mckinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 117–128 (2000)
21. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 84–98. Springer, Heidelberg (1986)
22. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43, 20–74 (1996)
23. Chico de Guzmán, P., Carro, M., Hermenegildo, M.V., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 197–213. Springer, Heidelberg (2008)
24. Marques, R.: Concurrent Tabling: Algorithms and Implementation. PhD thesis, Department of Computer Science, New University of Lisbon (2007)
25. Marques, R., Swift, T., Cunha, J.: Extending tabled logic programming with multithreading: A systems perspective (2008), `http://www.cs.sunysb.edu/~tswift`
26. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M.V., López, P., Puebla, G.: (Ciao Prolog System Manual), `http://clip.dia.fi.upm.es/Software/Ciao`
27. Madeira, S.C., Oliveira, A.L.: Biclustering algorithms for biological data analysis: a survey. IEEE/ACM Transactions on Computational Biology and Bioinformatics 1, 24–45 (2004)
28. Mechelen, I.V., Bock, H.H., Boeck, P.D.: Two-mode clustering methods: a structured overview. Statistical Methods in Medical Research 13, 979–981 (2004)
29. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge (2001)
30. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Data Mining, Inference and Prediction. Springer Series in Statistics (2001)
31. Moura, P., Crocker, P., Nunes, P.: High-Level Multi-threading Programming in Logtalk. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 265–281. Springer, Heidelberg (2008)
32. Moura, P., Rocha, R., Madeira, S.C.: Thread-Based Competitive Or-Parallelism. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 713–717. Springer, Heidelberg (2008)