Thread-Based Competitive Or-Parallelism*

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 subgoals compete for providing an answer and the first successful subgoal leads to the termination of the remaining ones. We discuss the implementation of competitive or-parallelism in the context of Logtalk, an object-oriented logic programming language, and present experimental results.

1 Introduction

Or-parallelism is a simple form of parallelism in logic programs, where the bodies of alternative clauses for the same goal are executed concurrently. Or-parallelism is often explored *implicitly*, without input from the programmer to express or manage parallelism. 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 [1] with committed-choice nondeterminism [2] and speculative threading [3]. 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. 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 one to

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

specify alternative procedures to solve a problem without caring about the details of speculative execution and thread handling. Another important key point of thread-based competitive or-parallelism is its simplicity and implementation portability when compared with classical, low-level or-parallelism implementations. The ISO Prolog multi-threading standardization proposal [4] 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 [5] are no longer available, due to the complexity of maintaining and porting their implementations.

Our competitive or-parallelism research is driven by the increasing availability of multi-core personal computing systems. These systems are turning into a viable high-performance, low-cost 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.

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.

Consider, for example, the *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. Assume now that we have implemented several methods to solve this problem, e.g. breadth-first, depth-first, and hill-climbing. In Logtalk, we may then write:

```
solve(Jugs, Moves) :-
   threaded((
      breadth_first::solve(Jugs, Moves)
   ;  depth_first::solve(Jugs, Moves)
   ;  hill_climbing::solve(Jugs, Moves)
   )).
```

The semantics of a competitive or-parallelism call implemented by the Logtalk built-in predicate `threaded/1` is simple. Given a disjunction of subgoals, a competitive or-parallelism call blocks until 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 throws an exception. The competitive or-parallelism call is deterministic and opaque to cuts; there is

no backtracking over completed calls. 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.

3 Implementation

In this section, we discuss the Logtalk [6] implementation of competitive or-parallelism, based on the core predicates found on the ISO standardization proposal for Prolog threads [4]. Logtalk is an open source object-oriented logic programming language that can use most Prolog systems 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 worrying about the details of thread management. Using Prolog core multi-threading predicates to support competitive or-parallelism allows simple and portable implementations to be written. Nevertheless, three major problems must be addressed when implementing or-parallelism systems: (i) multiple binding representation, (ii) work scheduling, and (iii) predicate side-effects.

Multiple Binding Representation A significant implementation advantage of competitive or-parallelism is that only the first successful subgoal in a disjunction of subgoals can lead to the instantiation of variables in the original call. This greatly 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`. Threads created with this predicate run a *copy* of the goal argument using its own set of data areas (stack, heap, trail, etc). Its implementation is akin to the environment copying approach [7], but much simpler as only the goal is copied. Because it is running a copy, no variable is shared between 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.

Work Scheduling Unrestricted competitive or-parallelism can lead to complex load balancing problems, since the number of running threads may easily exceed the number of available computational units. In our implementation, load balancing is currently delegated to the operating system thread scheduler. This is partially a consequence of our use of the core Prolog multi-threading predicates. However, and although we have postponed working on an advanced, high-level scheduler, 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 programming examples.

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 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 private. 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. Thread shared dynamic predicates are required by the ISO Prolog multi-threading standardization proposal to follow *logical update semantics*.

4 Experimental Results

In order to validate our implementation, we used competitive or-parallelism (COP) to simultaneously explore depth-first (DF), breadth-first (BF), and hill-climbing (HC) search strategies for the *water jugs* problem. 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 running Fedora Core 8 64 bits.[5]

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, BF, and HC 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 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 jugs problem. Moreover, given that we do not know *a priori* which individual 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. In particular, the initial thread data area sizes and the amount of memory that must be reclaimed when a thread terminates play a significant role on observed overheads. 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 the ISO Prolog multi-threrading standardization proposal. Nevertheless, even with the current implementations, our preliminary experimental results are promising.

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

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

5 Conclusions and Future Work

We have presented the logic programming concept of thread-based competitive or-parallelism supported by an implementation in the object-oriented logic programing language Logtalk. 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. Future work will include exploring the role of tabling in competitive or-parallelism calls and implementing a load-balancing mechanism. We also plan to apply competitive or-parallelism to non-trivial problems, seeking real-world experimental results allowing us to improve and expand our current implementation.

References

1. Ertel, W.: Performance Analysis of Competitive Or-Parallel Theorem Proving. Technical report fki-162-91, Technische Universität München (1991)
2. Shapiro, E.: The Family of Concurrent Logic Programming Languages. ACM Computing Surveys **21**(3) (1989) 413–510
3. González, A.: Speculative Threading: Creating New Methods of Thread-Level Parallelization. Technology@Intel Magazine (2005)
4. Moura, P.: ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Support Available from `http://logtalk.org/plstd/threads.pdf`.
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) (2001) 472–602
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**(2) (1990) 129–162