2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Scheduling OR-parallelism in YapOr and ThOr on Multi-Core Machines

Inês Dutra, Ricardo Rocha, Vítor Santos Costa, Fernando Silva and João Santos
CRACS & INESC-Porto TEC, Department of Computer Science, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
email:{ines,ricroc,vsc,fds,jsantos}@dcc.fc.up.pt

***Abstract*—In this work we perform a detailed study of different or-scheduling strategies varying several parameters in two or-parallel systems, YapOr and ThOr, running on multi-core machines. Our results show that some kinds of applications are sensitive to the choice of scheduling strategy adopted. In particular, the choice of scheduling parameters mostly affect applications that have short execution times, which, despite having speedups, have their performance significantly affected. Our results also show that topmost dispatching can be more advantageous than bottommost dispatching, a finding that contradicts previous works in this area. One last finding is that YapOr and ThOr are affected differently by changes in scheduling with ThOr performing significantly better than YapOr in several applications.**

***Keywords*-Scheduling strategies, Parallelism, Prolog implementation.**

I. Introduction

The last years have seen wide availability of multicore platforms, ensuring renewed interest in parallel programming. The difficulties inherent to parallel programming have motivated work in parallel execution of high-level declarative languages, both functional [1] and logic-based [2]. Declarative languages have two main advantages for parallelism. First, they allow high-level control of parallel execution. For example, the Prolog logic programming language has been extended to allow easy interprocess communication [3], protecting users from the nitty-gritty of setting up and managing parallel processes.

A second advantage of declarative systems is that they also offer the promise of *implicit* parallelism, as the semantics of a declarative program do not assume a sequence of execution steps. For instance, parallel logic programming systems have implemented two major forms of implicit parallelism: *and-parallelism* and *or-parallelism*. And-parallelism takes place when two goals are run independently. It often corresponds to divide-and-conquer or to producer-consumer parallelism, and its implementation poses challenging problems, such as granularity control, sharing of variables by different goals, and the need to perform parallel garbage collection. Or-parallelism takes place when there are several different alternative ways of solving the same goal. It is often found in applications that perform search. Or-parallelism is easier to implement, since different solutions for each goal (or-branches) can be searched quite independently.

Or-parallelism is present in many application areas such as parsing, optimization problems, and databases, and has been used to program at large by several companies [4], [5], [6], [7]. In contrast to and-parallelism, or-parallelism can be often exploited without the need of user generated annotations in the code. At the same time, its execution model is very close to the sequential one. This helps reuse of sequential implementation techniques. Indeed, efficient implementations of or-parallelism use the *multi-sequential* model. This approach is characterized by having a set of processes, that we shall name *workers*, where each one runs a full Prolog. In other words, each worker can solve a complete branch of the search tree.

The fundamental technical problem to solve in an or-parallel logic programming system is how to maintain multiple bindings for the same variable in different branches of the search tree [8]. In theory, a new *resolvent* is generated every time a goal is matched against the head of a clause, by renaming all the free variables of the current goal. In practice, such a copying process would be extremely costly. Instead, sequential Prolog systems simply rewrite the variables of the current goal in-place during the matching process. The conditional bindings are annotated in a special stack called *Trail* and will be undone later on backtracking.

Or-parallelism arises when the same goal matches several different clauses. Each clause may produce its own different set of bindings, and hence a system that allows the several clauses to proceed in parallel needs a mechanism to associate bindings to branches. Each processor needs to have its own copy of the conditional binding, as processing of each branch can be done separately and simultaneously with other processors. Several solutions have been proposed to cope with this problem. One of the frequently used classification considers three major groups: copying, sharing and recomputation [9]. Copying and sharing of stacks are the two most successful approaches.

Assuming that we have a working binding model, a second major decision in or-parallel systems is how to manage the highly irregular parallelism. The *or-scheduler* is responsible for task switching and has the function of choosing the best choice-point from which to take an alternative to be explored. As usual, a scheduler should not incur too high costs in task switching, and it should be able to cope with a limited number of processors in the system. Ideally, it should

978-0-7695-4676-6/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPSW.2012.199

reduce to the minimum the amount of interaction between workers. Some schedulers also deal with the problem of *speculativeness*. This means that whereas some branches of the search tree are to be executed compulsorily, others (speculative or-branches) might be pruned because of a cut or commit operator.

Several schedulers have been proposed and used in shared memory systems implementations such as Aurora [10], and Muse [11], that are close to modern multi-core architecture. Decisions the scheduler has to make include:

- *When to make work (or-branches) available, or public*? Making work public too late may result in idle workers; making it too soon may result in fine-grained parallelism and increase overheads.
- *Where to make idle workers go?* Should they take work close-by, or should they look at the root of the search–tree?
- *Should workers be proactive or lazy?* Lazy workers will hope for new work to appear near-by, so they may lose interesting work. On the other hand, too many proactive workers may lead to contention in the tree.

The literature shows significant discussion on this subject [12], [13], [14], with systems such as Aurora eventually supporting five different schedulers. In contrast, Muse followed a very tightly integrated design, based on distributing work bottom-up. The approach was followed in the YapOr system [15], with some flexibility in allowing some decisions to be user-controlled. Recent work [16] showed that YapOr was still effective in modern multi-core systems. Moreover, it showed that the process-based design of YapOr could be transformed into an effective thread-based design, ThOr.

In this work we investigate the issues in or-scheduling strategies for YapOr and ThOr. Both ThOr and YapOr share the original YapOr scheduler, itself based on decisions taken by the Muse and Aurora developers. We address two main questions. First, we investigate whether the schedulers are *effective*: are we close to optimal performance, or do we need significant work to make or-schedulers effective in novel machines with very different hardware. Second, we study *sensitivity*: how do the differences between YapOr and Thor affect performance, and whether we need substantial tuning effort.

Our work shows that YapOr and Thor seem indeed to perform quite well in terms of exploiting available work, and that are not very sensitive to, nor do they require much, parameter tuning. On the other hand, we show that the best scheduling decision may depend on the type of application and on the number of available cores. Last, although performance of YapOr and Thor is similar, as expected, they do react in different ways to differences in scheduling.

The remainder of the paper is organized as follows. First, we briefly introduce the YapOr and ThOr models and describe their main data structures, memory organization and default scheduling strategies. Then, we discuss the main or-scheduler parameters and different values that they can take. Next, we present the benchmarks used in this study, show our results and discuss the impact of the different strategies and parameters in the applications. Finally, we present our conclusions and draw perspectives of future work.

II. The YapOr Model

The initial implementation of or-parallelism in YapOr was largely based on the *stack copying model* as first introduced by Ali and Karlsson in the Muse system [17], [18]. YapOr is an example of a *multi-sequential* model [19], where each processor or *worker* maintains its own copy of the search tree. Only when a worker runs out of work, it searches for work from fellow workers. If a fellow worker has work, it can make some or all of its open alternatives available: this operation is called *sharing*. First, the sharer will make some or all of its choice-points *public*, so that backtracking to these choice-points can be synchronized between different workers. Second, in a copying model, the execution stacks of the sharer are copied to the requester. The sharer then continues forward execution, while the requester backtracks to the shared choice-points and exploits alternatives.

YapOr follows a scheduling strategy based on bottommost dispatching of work. Both, for stack copying and stack sharing, this was shown to be more efficient than topmost strategies [18], [13]. Synchronization between workers is mainly done through choice-points. In a copying model, each worker has a separate copy of the public choice-points. Synchronization requires an auxiliary data structure, called *or-frame*, to be associated with the public choice-points. The or-frames form a tree that represents the public search tree.

Sharing work is a major source of overhead in YapOr, as it requires copying the execution stacks between workers. The *incremental copying strategy* [17] is designed to reduce this overhead by allowing the receiving worker to keep those parts of its execution stacks that are consistent with the giving worker, and only to copy the differences between the two workers' stacks.

The YapOr memory is divided into two major *shared* address spaces: the *global space* and a collection of *local spaces*. The global space contains the code area inherited from Yap and all the data structures necessary to support parallelism. Each local space represents one system worker and contains the four execution stacks inherited from Yap: heap, local, trail, and auxiliary stack.

In order to efficiently meet the requirements of incremental copy, the set of local spaces are mapped as follows. The starting worker asks for shared memory in the system's initialization phase. Afterward, the remaining workers are created and inherit the previously mapped address space. Then, each new worker rotates the local spaces, in such a way that all workers will see their own spaces at the same virtual memory addresses.

III. The ThOr Model

The ThOr model builds upon two major components of Yap: the YapOr implementation, as discussed in the previous section, and the Yap threads library [20]. The Yap thread library can be seen as a high-level interface to the POSIX threads library, where each thread runs on a separate stack but shares access to the global data structures (code area, atom table and predicate table). As each thread operates its own stack, thus, it is natural to *associate* each parallel worker to a thread: threads can run in parallel and they already include the machinery to support shared access and updates to the global data structures and input/output structures.

Notice however that not all threads have to be workers: some threads may be used to support specialized tasks, possibly running in parallel with the workers. We believe this is an important advantage of ThOr. Traditionally, we expect to find a single *or-parallel* program, and issues such as side-effects must be addressed within the or-parallel system. In ThOr, we can easily construct independent threads that *collaborate* or even *control* the workers (or-parallel threads). Natural applications are the collection of solutions, and performing input/output tasks.

As YapOr, the memory model in ThOr is also based on stack copying. Thor uses copying for several reasons. First, copying allows us to preserve a key notion in the thread library: independent and separate workers have a *private* stack. In this way, we can reuse the existing code for threads so that workers can independently perform garbage collection and stack shifting. Second, as workers see a contiguous stack, copying imposes less overheads on the engine and has higher performance compared with other approaches [21]. Ultimately copying is less intrusive on the sequential engines than other approaches. As a small experiment, we evaluated what kind of changes would be needed in the emulator. In both models, support for or-parallelism, including copying, requires about 60 changes to the emulator, mostly in order to adapt choice-point manipulation and to perform pruning on the shared tree. Support for a model based on SBA (Shared Binding Arrays)requires 90 additional changes that affect a complex operation, unification. On the other hand, YapOr's copying model relies on every worker *having its own stacks at the same virtual address position*. This clearly will not work with threads.

In a thread-based model, all memory areas should be visible to all threads *at the same virtual memory address positions*. Moreover, in order to take full advantage of memory (especially on 32 bit machines), it is convenient not to assume any preconditions on memory organization. In Yap, threads may actually move their stacks in the virtual memory space during execution.

ThOr has been designed to take advantage of the existing YapOr code-base but, having each stack at the same virtual memory addresses, does not hold true in ThOr. Namely, to share work we need to have several copies of the same choice-point at different virtual memory addresses. In ThOr, to address this problem, our key idea is *shifted copying*, which essentially consists of two steps: copy the memory between the workers sharing work and then adjust pointers. Although shifted copying adds a linear overhead to copying operations, it offers some important advantages: (1) it allows using the thread infra-structure as is, and (2) it allows shifting between stacks *with different sizes*, and we can actually reuse preexisting code from the Yap stack shifter.

IV. Default Scheduling Strategy

When a worker runs out of work, the scheduler tries to find a busy worker with excess of work load to share work. There are two alternatives to search for busy workers in the search tree: search *below* or search *above* the current node. In YapOr idle workers always start to search below the current node, and only if they do not find any busy worker there, then they search above. The main advantage of selecting a busy worker below instead of above is that the idle worker can immediately perform the sharing operation, because its current node is already common to the busy worker, thus avoiding the need to backtrack in the tree until finding a common ancestor.

When the scheduler does not find any busy worker with excess of work load, it tries to move the idle worker to a better position in the search tree. By default, the idle worker backtracks until it reaches a node where there is at least one busy worker below. Another option is to backtrack until reaching the node that contains all the busy workers below. The goal of these strategies is to distribute the idle workers in such a way that the probability of finding, as soon as possible, busy workers with excess of work below is substantially increased.

Busy workers may choose to make their work public. In the YapOr and ThOr implementations, workers make their work public whenever their load exceeds a threshold. This is parametrized. Each worker annotates its load (total accumulated number of alternatives in its branch) at each choice-point. After following the default search order in the code, idle workers have a choice of hanging around for a while and wait for work to become available. This is controlled by a second scheduler parameter. We will study in this work whether the order of searching for work and changes in parameter values can affect the performance of or-parallel applications.

V. Methodology and Experimental Results

Our performance evaluation focuses on answering the question: up to which extent different scheduling strategies and choices of scheduling parameters affect the parallel execution of Prolog programs?

We would need to consider an enormous number of possible combinations in order to fully answer this question.

In this study, we chose to first explore different search strategies, while using default parameter values, and obtain a performance trend. Later, using the best choice of search order, we vary the parameter values for the search. It turns out that the choice of an order for the search is not straightforward and indeed depends on the number of processors used and on the application. Therefore, we performed experiments for three different search orders with default parameter values, and using the default order of searching for work, we varied the parameter values. Whenever we found a trend, we performed complementary experiments varying parameters for search orders different from the default.

The following programs were used as benchmarks:

- `cubes`: a known benchmark taken from Tick's book [22]. This program implements the solution of a magic cube of size 7.
- `ham`: checks if a given graph forms a Hamiltonian cycle. The graph instance used here has 26 nodes.
- `magic`: a solution to the 3x3x3 magic cube.
- `map` and `mapbigger`: a solution for the map coloring problem using four colors. We used two maps representing diverse size and graph density. The smaller version has 15 nodes and the bigger version has 17 nodes.
- `puzzle`: one version of sudoku where the diagonals must add up to the same amount.
- `puzzle4x4`: a solution for a 4x4 maze.
- `queens`: a solution for the n-queens problem using forward checking. The size of the board used in testing was 13x13.

First, we studied the execution patterns of each application:

- `cubes`: generates a top-level sequence choice-point with 7 alternatives, and each alternative creates a choice-point with four alternatives. The process repeats itself on each branch. The maximum depth of the tree is 7.
- `ham`: the choice-points in this program always have only two alternatives. The depth of the tree is 26, with a choice-point of size 2 created at each level.
- `magic`: the first choice-point created has ten alternatives. Each alternative recursively creates another choice-point of size 10. The maximum depth of the tree is 7.
- `map`: the choice-points in this application have only two alternatives. The minimum depth of the tree is 15 and the maximum is limited by the maximum number of neighbors a country can have. As the maximum number of neighbors for this problem in this study is five, the maximum depth is 20. For each level in a branch, a two-alternative choice-point is created.
- `mapbigger`: it has the same execution pattern of map, but a larger tree, with a minimum depth of 17 and maximum, 22.
- `puzzle`: this has a computational pattern that starts with a choice-point with 19 alternatives. For each branch, new choice-points of consecutive sizes 18 down to 12 are created.
- `puzzle4x4` this tree has maximum depth of 12. At each level of the tree choice-points are created, whose number of alternatives vary from one to 48.
- `queens` this instance of n-queens creates an initial choice-point of size 13, with each branch recursively creating successive choice-points of size two.

These programs have been used in previous work; most of them obtained very good performance up to 24 processors with versions of YapOr and ThOr that used the default search order and the default parameter settings [16].

We performed our experiments on a dual Six-Core AMD Opteron(tm) Processor 2435, with 64 GBytes of RAM, running Red Hat Enterprise Linux in 64-bit mode. Each experiment was run 30 times and results are the average of the 30 runs reported in seconds. We ran each experiment using one, two, four and eight processors. Statistical significance tests were applied to compare average execution times. We use the two-tailed t-test for the statistical significance tests. Whenever applicable, we report the *p-values*. In the t-test, the closest the p-value is to zero, the greater the probability of refusing the null hypothesis that states that the two algorithms being compared are the same.

We perform three types of experiments. First, we experiment with varying the order used by workers when searching for available work. to publish work. We follow three strategies: bottom-most first, the default strategy; top-most work first; and best work in the tree first.

Second, we experimented with varying how eager a worker is to search for work. To control how a worker searches for work we use the parameter sl, where sl is the number of times the worker must loop before looking for hidden shared work. The default parameter values for sl is 10. We varied sl around this value with sl taking values 8, 6, 4 and 12, 14, 16, 18;

Last, we experimented with varying the amount of work shareable in the tree. The d parameter is the load threshold above which an idle worker can claim work from a peer. This parameter works as a delay in getting work, since an idle work hangs waiting that some busy worker publishes its excess work. The default setting for d is 3, and we vary d to take the values 2, 1, 0 and 4, 5, 6, 7.

A. Performance of YapOr and ThOr, varying the search order

Our first set of experiments addresses the question of scheduling strategy: what is the impact of using different scheduling strategies, and how does it vary between application and model? To study this problem, we vary the order

of searching for work and actually create three different versions of YapOr and ThOr. We called the first version *BELOW*. It was compiled to follow the default order search order: first, look for work below; if work is not found below, look for work above; if work is not found above, look for a better position in the search tree (this strategy was explained in Section IV). The second version was called *ABOVE*. It was compiled to search first for work above, and if no such work is available, to search for work below and then look for a better position. The third version was called *BETTER*. It was compiled to first find a better position in the tree, then look for work below and then look for work above.

Results for the first set of experiments are shown in Figures 1 and 2. For each application, the X-axis shows the number of workers (2, 4 and 8), and the Y-axis shows average execution times in seconds. Notice that we do not show execution times for a single CPU. Each graph shows the performance of an application either using YapOr or ThOr and three curves representing the three ways of searching for work (below, above and finding a better position). YapOr and ThOr are always run with default parameters.

In Figure 1, we show smaller applications. They run from 0.14 to 7.31 seconds on 1 processor. In Figure 2, we show applications with the longest run-times, that run from 19 to 52 seconds on a single processor. The first important conclusion we can take from these pictures is that, even using mechanisms that used to work in old shared memory machines, YapOr can still achieve very good speedups. This is especially the case for applications with longer run-times, with some applications reaching linear or quasi-linear speedups.

The results show that with two workers some applications are significantly affected by the way workers search for work, with a variation of almost 4s in 30s (`mapbigger`, YapOr, two workers). This effect holds true even in applications with long running-times, and is less noticeable for larger configurations, four or eight workers.

For two workers, the p-value is less than .0008 for most of the comparisons between below-above and below-better. As mentioned before, such a low values of p strongly suggests a statistical significant difference between two strategies. The overall best order of search for this number of processors to search for work *above*. This is not true for every application, even at two workers, the application `cubes` does not seem to be affected by a change from searching first for work below and searching for work from above ($p = .076$), but it is affected if we change the search to look for a better location in the tree ($p < .0009$). The application `ham`, running with YapOr, two workers, is affected by a change from seeking for work below to seeking for a better location ($p = .11$). The application `puzzle` also is not affected by a change from seeking for work below to seeking for a better location, for both YapOr and ThOr, two workers ($p = .53$ for YapOr and $p = .13$ for ThOr).

In the case of two workers, it is interesting to notice that all, but one application, have higher average execution times with the version of YapOr that looks for work below (bottom-most dispatching). The same holds true for most of the ThOr executions. This contradicts previous works, in particular the ones related to the Muse system (whose implementation is the basis of YapOr), that reported bottom-most dispatching as the better search strategy. In modern machines the access to memory is faster than in older shared memory machines. Therefore, bottom-most dispatching may not be a good choice because of memory contention when more than one worker needs to access the same region of the Prolog stack.

YapOr and ThOr perform different for four workers. The applications that present statistically different execution times using ThOr are `ham` and `puzzle` ($p = .008$ comparing below with better) and `puzzle4x4` ($p = .0005$ comparing below with better). Both have better average execution times using search for a better location instead of searching for bottom-most work. The exception is the `map` application, that has a statistically significant difference between below and better ($p = .0009$), with below having the best average execution time. In practice, the more workers we have, the more contention. Therefore, dispatching on topmost or looking for a better location prevents workers from competing for the same choice-point. This is particularly true for the applications mentioned, because they have many choice-points left on the topmost part of the tree.

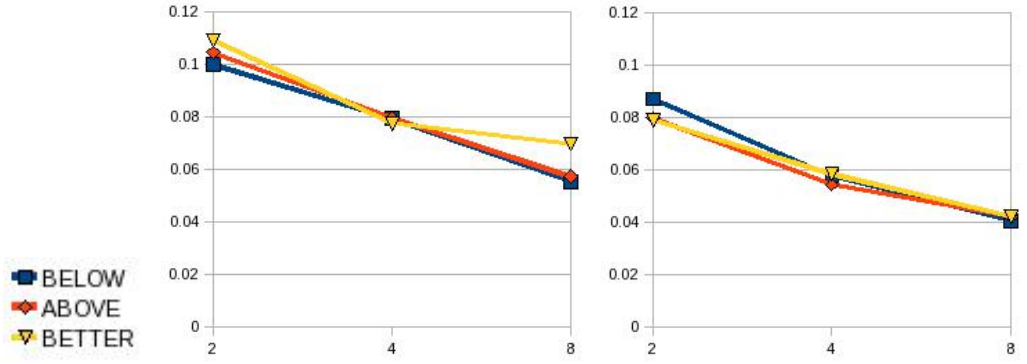
For YapOr, four workers, the only statistically significant difference is found for `mapbigger` with bottom-most dispatching being the best strategy. We believe that this difference of behavior between YapOr and ThOr is related to the memory organization of both systems. While ThOr is thread-based and has to manipulate a single page table, YapOr overheads associated to the maintenance of the several page tables per process. Therefore, moving to topmost parts of the tree in ThOr can be beneficial.

Except for `ham`, `puzzle`, `puzzle4x4` and `map`, in the cases mentioned, all the other results are not statistically significant, indicating that any order of search could be used for the remaining applications.

For eight workers, results show again that the default search order is not always the best strategy. For the short-running applications (Figure 1), we can see a statistically significant difference for `cubes` and `ham`, running with YapOr. On the other hand, the ThOr executions with eight workers are not significantly affected by a change in the search order. For longer-running applications (Figure 2), there are no statistically significant differences among the three strategies: they all perform quite well.

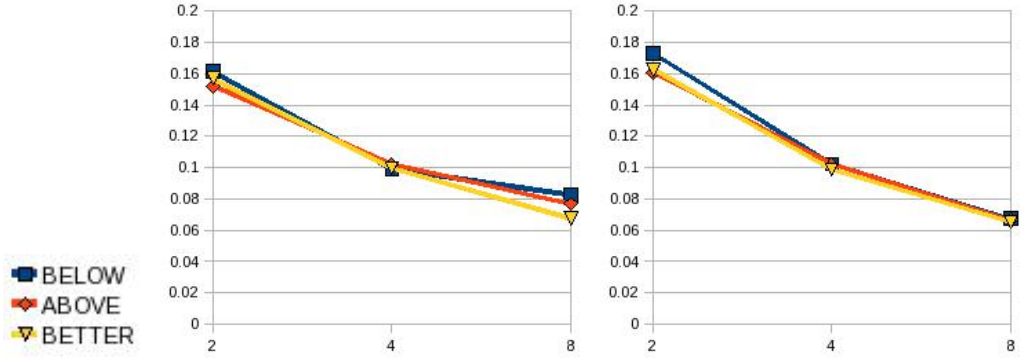
Table I summarizes the best search order for different numbers of processors for all applications. The value "any" corresponds to non significant differences among the search

Figure 1. Varying the search order in `cubes`, `ham`, `puzzle` and `puzzle4x4`. The X-axis shows the number of workers (2, 4 and 8), and the Y-axis shows average execution times in seconds.



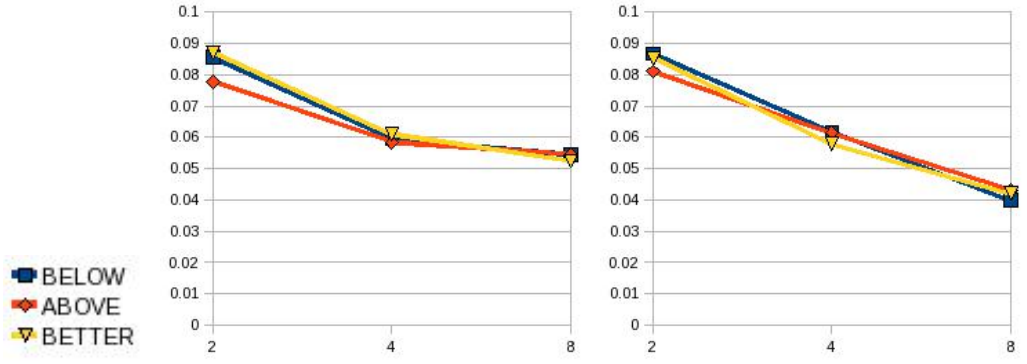
(a) `cubes, YapOr`

(b) `cubes, ThOr`



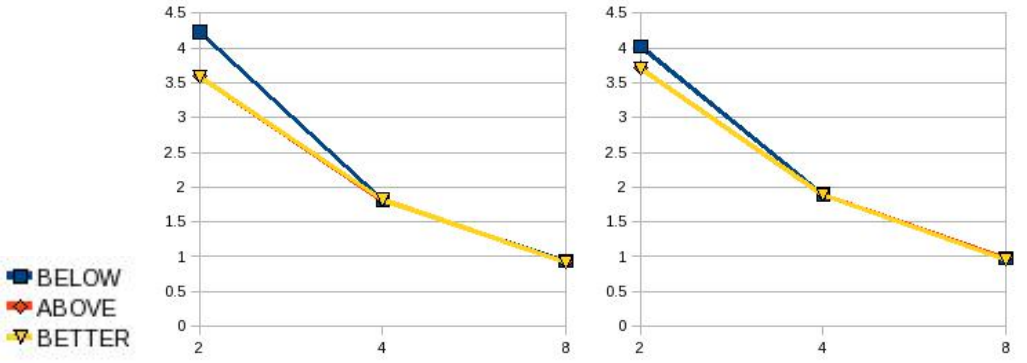
(c) `ham, YapOr`

(d) `ham, ThOr`



(e) `puzzle, YapOr`

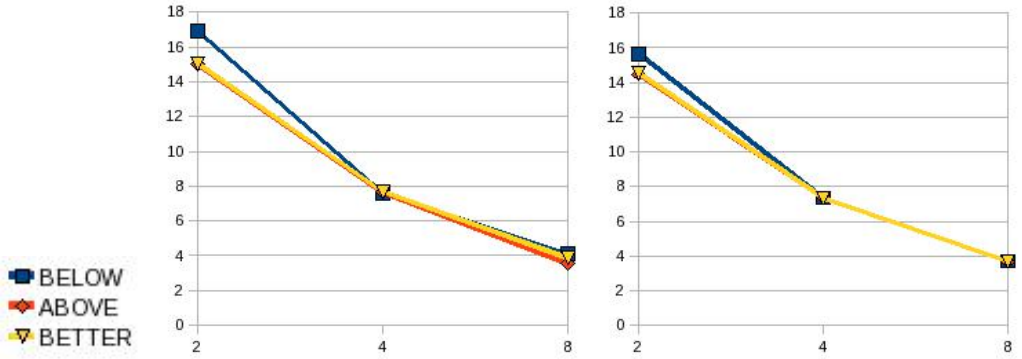
(f) `puzzle, ThOr`



(g) `puzzle4x4, YapOr`

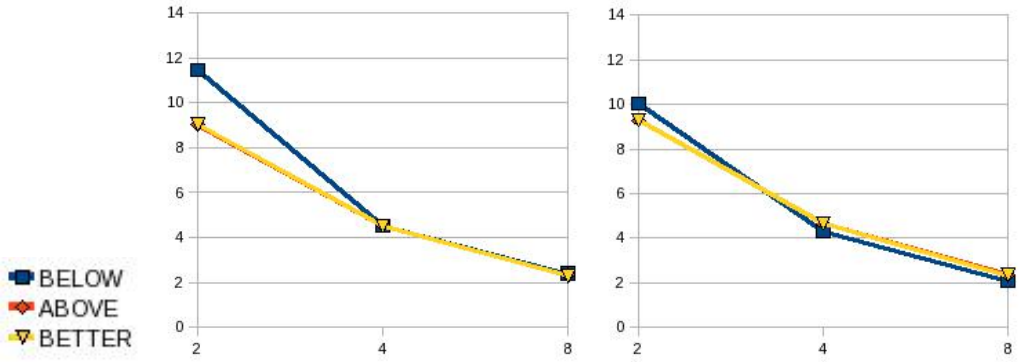
(h) `puzzle4x4, ThOr`

Figure 2. Varying the search order for `magic`, `map`, `mapbigger`, and `queens`. The X-axis shows the number of workers (2, 4 and 8), and the Y-axis shows average execution times in seconds.



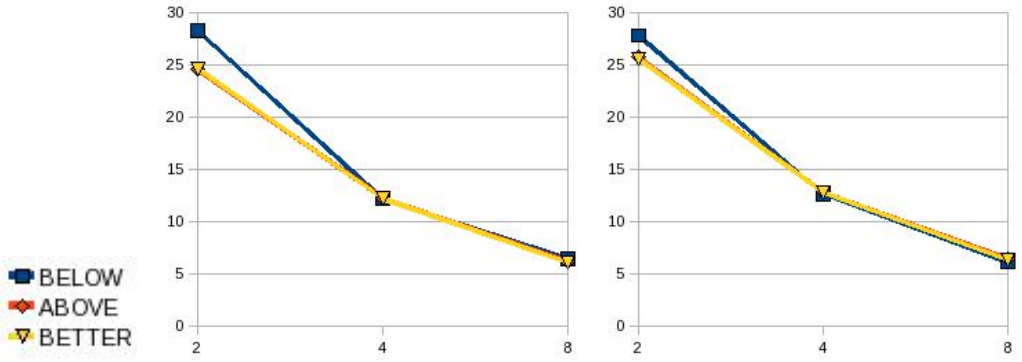
(a) `magic, YapOr`

(b) `magic, ThOr`



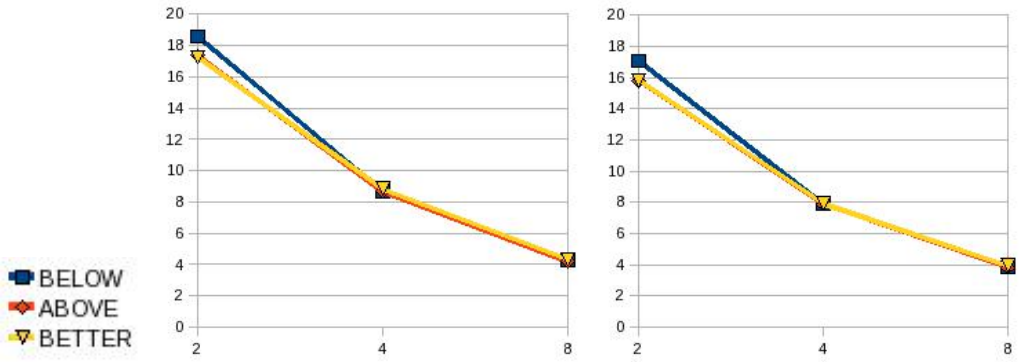
(c) `map, YapOr`

(d) `map, ThOr`



(e) `mapbigger, YapOr`

(f) `mapbigger, ThOr`



(g) `queens, YapOr`

(h) `queens, ThOr`

orders. All the other values are significantly different with $p < .00823$, with almost all *p-values* equal to zero. For two workers, as mentioned before, search for work above favors performance in both YapOr and ThOr.

Table I shows almost no statistical difference among the search orders for four workers. For eight workers the statistical difference reappears, with YapOr being more affected by a change of search order than ThOr.

B. Performance of YapOr and ThOr, varying parameters

In this set of experiments, we keep the default order of search for work (look for work below) and vary parameter values for the search.

Figure 3 shows the variation in average execution times (for 30 runs, per application, per parameter setting). as we increase the number of workers. The applications are `mappbigger`, `queen`, `map`, `puzzle4x4`, `ham`, and `puzzle`. We organize the graph by pairs sl, d. Notice also that the graph shows absolute running times.

The results indicate that the choice of parameter values affects performance, especially under ThOr. For two workers, the *p-value* for `cubes`, when comparing the performance of YapOr using parameter settings 6-0 and 6-7 ($sl = 6$, $d = 0$ and $sl = 6$, $d = 7$, the ones that produced the lowest average execution times) is 0.000032. This value confirms that the different settings of parameters definitely affect performance with a high statistical significance (with confidence close to 100%). For two workers, it seems that the dominant parameter setting for most of the applications is the combination of $sl = 6$ and $d = 0$ (the setting that produces the lowest average execution times) indicating that the values of sl and d, used as default by YapOr and ThOr, may be overestimated and underestimated, respectively. More precisely, compared with the default parameters ($sl = 10$ and $d = 3$), this means that it is better to look for work earlier instead of being too conservative, and it is better not to delay the sharing of work.

We performed an additional set of experiments, where we vary the sl and d parameters for two, four and eight workers, using search for work above, because it showed a good performance with default parameters for almost all applications with two workers. Results for this experiment can be found in Figure 3. Further results, not presented here for lack of space, show that YapOr and ThOr consistently perform better with the order of search above than with the default search order.

The results in Figure 3, show that for applications like `mapbigger` and `map`, which have a rather deep tree with at most two alternatives per checkpoint, and with high average execution times, the variation of parameters makes the execution very unstable for ThOr. However for all combinations of parameters, ThOr can execute faster than YapOr, although YapOr is less affected by a change in parameter values.

VI. Conclusions and Future Work

In this work, we studied how or-scheduling strategies implemented in or-parallel Prolog systems affect the performance of Prolog applications in modern multi-core architectures. We performed a thorough study, where we vary: (1) the main strategies to search for work: bottom-most dispatching, top-most dispatching and looking for better work, and (2) parameter values to control when looking for work and sharing available work. Our main conclusions are: (1) contrary to what is discussed in the literature, a choice of top-most dispatching or looking for better work can produce statistically significant better results than dispatching on bottom-most work; (2) ThOr is more sensitive to a change on parameter values for a larger number of processors; (3) YapOr is more sensitive to a change on searching strategy order; (4) applications that take longer to execute, with deep depth and few alternatives per choice-point can run faster in Thor for a large number of processors, but at the cost of being less stable to a variation on parameter values.

This work uncovered several interesting issues not yet investigated in or-parallel Prolog systems. Work is being carried out to delve more deeply into these differences between YapOr and Thor. Ultimately, our results will contribute to our goal of dynamically deciding what is the best search strategy and better combination of parameters, based on architectural and application parameters.

Acknowledgments

This work has been partially supported by the FCT research projects LEAP (PTDC/EIA-CCO/112158/2009) and HORUS (PTDC/ EIA-EIA/100897/2008).

References

[1] K. Hammond and G. Michelson, Eds., *Research Directions in Parallel Functional Programming*. London, UK: Springer-Verlag, 2000.

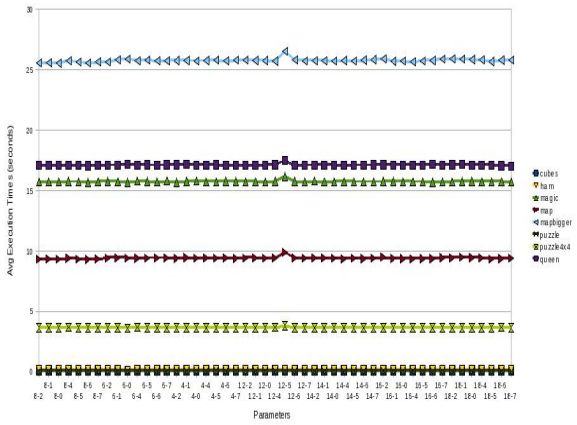
[2] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel Execution of Prolog Programs: A Survey," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, 2001.

[3] N. A. Fonseca, A. Srinivasan, F. M. A. Silva, and R. Camacho, "Parallel ILP for distributed-memory architectures," *Machine Learning*, vol. 74, no. 3, pp. 257–279, 2009.

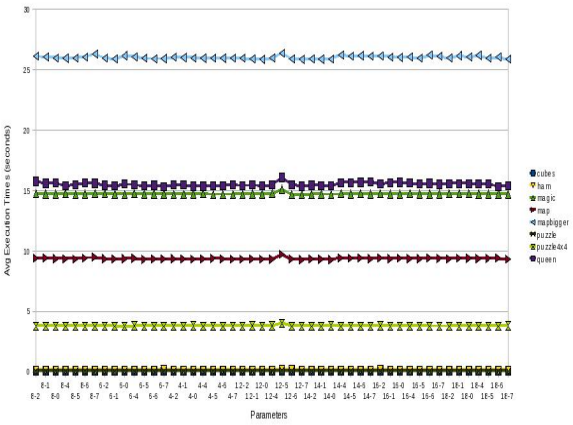
[4] M. Aref, "Logic for enterprise decision automation applications," in *Commercial Users of Logic Programming, in conjunction with ICLP, 2009*, 2009, company: LogicBlox.

[5] L. Spratt, "The ontology works deductive database system," in *Commercial Users of Logic Programming, in conjunction with ICLP, 2009*, 2009, company: Ontology Works Inc.

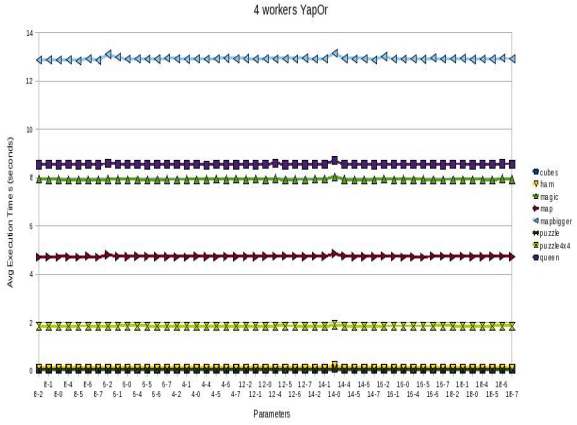
[6] D. Warren, "From data to knowledge," in *Commercial Users of Logic Programming, in conjunction with ICLP, 2009*, 2009, company: XSB Inc.



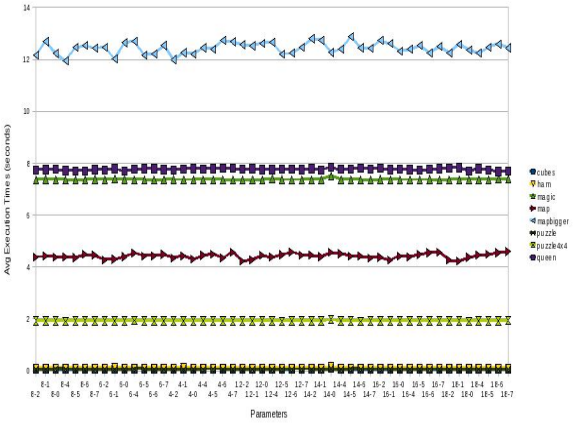
(i) `Two workers, YapOr`



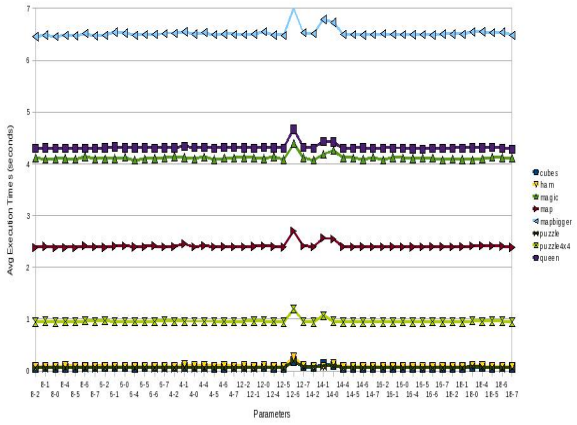
(j) `Two workers, ThOr`



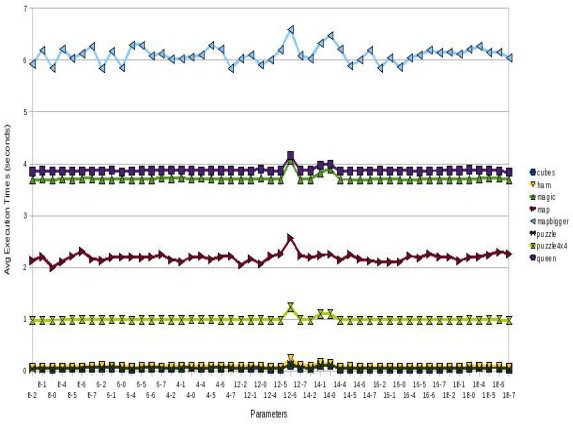
(k) `Two workers, YapOr`



(l) `Four workers, ThOr`



(m) `Eight workers, YapOr`



(n) `Eight workers, ThOr`

Figure 3. Varying parameters with search = above

Table I
BEST SEARCH ORDER

Benchmark	Two workers		Four workers		Eight workers	
	YapOr	ThOr	YapOr	ThOr	YapOr	ThOr
cubes	below/above	better	any	any	below	any
ham	above	above	any	better	better	any
magic	above	above	any	any	above	any
map	above	better	any	below	better	below
mapbigger	above	better	below	any	better	below
puzzle	above	above	any	better	any	any
puzzle4x4	better	better	any	better	above	better
queens	better	above	any	any	above	below/above

[7] M. Elston, "From prolog to porsche: experiences developing a large scale financial application in prolog," in *Commercial Users of Logic Programming, in conjunction with ICLP, 2009*, 2009, company: Scientific Software and Systems Ltd.

[8] J. A. Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages," in *Int. Conf. on Parallel Processing*, D. DeGroot, Ed. St. Charles, Ill.: IEEE, August 1985, pp. 131–138.

[9] J. C. Kergommeaux and P. Codognet, "Parallel logic programming systems," *Computing Surveys*, vol. 26, no. 3, pp. 295–336, September 1994.

[10] E. Lusk *et al.*, "The Aurora or-parallel Prolog system," *New Generation Computing*, vol. 7, no. 2,3, pp. 243–271, 1990.

[11] K. Ali and R. Karlsson, "The Muse Or-parallel Prolog Model and its Performance," in *Proceedings of the North American Conference on Logic Programming*. MIT Press, October 1990, pp. 757–776.

[12] ——, "Scheduling Speculative Work in MUSE and Performance Results," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 449–476, 1992.

[13] A. Beaumont, S. Raman, P. Szeredi, and D. H. D. Warren, "Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler," in *Conference on Parallel Architectures and Languages Europe*, ser. LNCS, no. 506. Springer-Verlag, 1991, pp. 403–420.

[14] R. Sindaha, "Branch-Level Scheduling in Aurora: The Dharma Scheduler," in *International Logic Programming Symposium*. The MIT Press, 1993, pp. 403–419.

[15] R. Rocha, F. M. A. Silva, and V. Santos Costa, "Yapor: an or-parallel prolog system based on environment copying." in *EPIA'99*, 1999, pp. 178–192.

[16] V. Santos Costa, I. Dutra, and R. Rocha, "Threads and or-parallelism unified," *Theory and Practice of Logic Programming, 26th International Conference on Logic Programming (ICLP 2010), Special Issue*, vol. 10, no. (4–6), pp. 417–432, July 2010.

[17] K. Ali and R. Karlsson, "The Muse Approach to OR-Parallel Prolog," *International Journal of Parallel Programming*, vol. 19, no. 2, pp. 129–162, 1990.

[18] ——, "Full Prolog and Scheduling OR-Parallelism in Muse," *International Journal of Parallel Programming*, vol. 19, no. 6, pp. 445–475, 1990.

[19] K. Ali, "Or-parallel Execution of Prolog on a Multi-Sequential Machine," *International Journal of Parallel Programming*, vol. 15, no. 3, pp. 189–214, 1986.

[20] V. Santos Costa, "On Supporting Parallelism in a Logic Programming System," in *Workshop on Declarative Aspects of Multicore Programming*, 2008, pp. 77–91.

[21] V. Santos Costa, R. Rocha, and F. M. A. Silva, "Novel Models for Or-Parallel Logic Programs: A Performance Analysis," in *EuroPar 2000 Parallel Processing*, ser. LNCS, no. 1900. Springer-Verlag, 2000, pp. 744–753.

[22] E. Tick, *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, 1987.