Achieving Scalability in Parallel Tabled Logic Programs

Ricardo Rocha, Fernando Silva
DCC-FC & LIACC
Universidade do Porto, Portugal
{ricroc,fds}@ncc.up.pt

Vítor Santos Costa*
COPPE Systems
Universidade do Rio de Janeiro, Brasil
vitor@cos.ufrj.br

Abstract

Tabling or memoing is a technique where one stores intermediate answers to a problem so that they can be reused in further calls. Tabling is of interest to logic programming because it addresses some of most significant weaknesses of Prolog. Namely, it can guarantee termination for programs with the bounded term-size property. Tabled programs exhibit a more complex execution mechanism than traditional Prolog's left-to-right search with backtracking. The reason is that Prolog programs are highly recursive and generate multiple answers. This rather involved execution mechanism requires a more complex implementation than traditional Prolog.

The declarative nature of tabled logic programming suggests that it might be amenable to parallel execution. On the other hand, the complexity of the tabling mechanism, and the existence of a shared resource, the table, argues that parallelism might be limited, and that performance for real applications might never scale. In this work we prove that parallel tabling is indeed scalable for real applications by experimenting the OPTYap parallel tabled system on a scalable shared-memory machine.

Keywords: *Parallel Logic Programming, Tabling.*

1. Introduction

Tabling or memoing [9] is a technique where one stores intermediate answers to a problem so that they can be reused in further calls. Memoing was originally introduced in the context of Artificial Intelligence, but has since been introduced to areas as diverse as computer architecture and the functional languages. The last few years have seen significant work in tabling from the logic programming community. The motivation for this work is that tabling addresses some of most significant weaknesses of Prolog. Namely, whereas Prolog is quite vulnerable to infinite loops, tabling can guarantee termination for all programs with the *bounded term-size property* [2]. Moreover, tabling can often significantly reduce the search space for logic programs.

Most of the ground-breaking work in tabling for logic programming has been developed by the XSB group, from a novel resolution strategy, SLG-resolution [1], to a new abstract machine, the SLG-WAM [17], which forms the basis for XSB Prolog system [7]. XSB has been used with success for applications such as natural language processing, knowledge-base systems and data-cleaning, and program-analysis. One application where XSB-based work has achieved remarkable results is in model-checking, through the XMC system [6].

Tabled programs exhibit a more complex execution mechanism than traditional Prolog's left-to-right search with backtracking. The reason is that Prolog programs are highly recursive and generate multiple answers. For tabling to be effective, SLG-resolution therefore does not recompute calls to variant tabled goals (consumers), even if the original goal, or producer, has not been fully computed yet. Hence, answers must be sent from producers to consumer as we find them. Of course, sending a answer to a consumer may in turn lead to a new answer for the producer! This rather involved execution mechanism requires a more complex implementation than traditional Prolog, whilst demanding declarative programming. The declarative nature of tabled logic programming suggests that it might be amenable to parallel execution, as it has been previously obtained for Prolog. On the other hand, the complexity of the tabling mechanism, and the existence of a shared resource, the table, argues that parallelism might be limited, and that performance for real applications might never scale.

In this work we present and study the performance of what to the best of our knowledge is the first parallel tabling logic programming system, OPTYap. OPTYap is based on the ground-breaking work in XSB, adapted to the high-performance Yap Prolog system [3]. Our key idea in exploiting parallelism is that quite a few interesting applica-

*This work has been partially supported by Fundação para a Ciência e Tecnologia, CLoP (CNPq) and PLAG (FAPERJ).

tions of tabling are by nature non-deterministic. We therefore argue that we should be able to run in parallel alternatives from both tabled and non-tabled goals. By doing so we can both extract more parallelism, and we can reuse the technology presented for or-parallelism (ORP) and tabling. In our case, we extended Yap to support *Or-Parallelism within Tabling (OPT)* [14]. The OPT model considers tabling as the base component of the system, that is, each computational worker behaves as a full sequential tabling engine. The ORP component of the system is triggered when a worker runs out of alternatives to exploit. The OPT model gives the highest degree of orthogonality between or-parallelism and tabling, thus simplifying initial implementation issues.

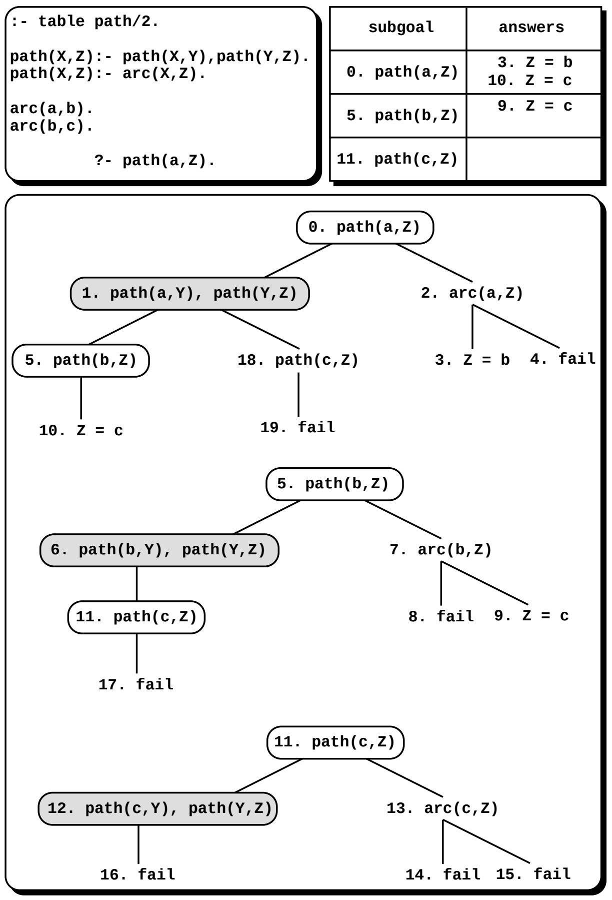
Our major goal is to study the scalability of the OPTYap system. We use several examples of the XMC model checking application, plus well-known tabled benchmarks, on a scalable shared-memory architecture, the SGI Origin2000. Our results show that scalability is indeed possible: linear speedups have been obtained on XMC up to 32 processors.

The remainder of the paper is organized as follows. First, we briefly introduce the basic ideas behind tabling in the context of logic programming. Next, we discuss the main issues in the implementation of OPTYap. We then characterize applications and present performance data. We terminate by outlining some conclusions and suggesting further work.

2. Basic Tabling Definitions

The basic idea behind tabling is straightforward: programs are evaluated by storing newly found answers of current subgoals in a proper data space, called the *table space*. The method then uses this table to verify for repeated calls to subgoals. Whenever such a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses. In the following, we illustrate the tabled evaluation through an example.

Consider the Prolog program of Figure 1 that defines a small directed graph (represented by the `arc/2` predicate) with a relation of reachability (given by the `path/2` predicate), and the query goal `?- path(a,Z)`. Traditional Prolog would enter an infinite loop because the first clause of `path/2` leads to a repeated call to `path(a,Z)`. In contrast, if tabling is applied then termination is ensured. Figure 1 illustrates the evaluation sequence when using tabling. At the top, the figure illustrates the program code and the state of the table space at the end of the evaluation. Declaration `:- table path/2` in the program code indicates that predicate `path/2` should be tabled. The bottom block shows the resulting forest of trees for the three tabled subgoal calls. The numbering of nodes denotes an evaluation sequence (several are possible).



```
:- table path/2.

path(X,Z):- path(X,Y),path(Y,Z).
path(X,Z):- arc(X,Z).

arc(a,b).
arc(b,c).

        ?- path(a,Z).
```

subgoal	answers
0. path(a,Z)	3. Z = b 10. Z = c
5. path(b,Z)	9. Z = c
11. path(c,Z)	

Figure 1. A finite tabled evaluation.

Whenever a tabled subgoal is first called, a new tree is added to the forest of trees and a new entry is added to the table space. We say this call corresponds to a *producer node*. In this case, execution starts with a producer node, node 0 (first call to `path(a,Z)`). The evaluation thus begins by creating a new tree rooted by `path(a,Z)` and by inserting a new entry in the table space for it. Next, `path(a,Z)` is resolved against the first clause for `path/2`, creating node 1.

Tabling is useful when we find repeated or *variant* calls to tabled subgoals. Node 1 is an example, where we found a variant of `path(a,Z)`. Other examples include node 6 and node 12. At such nodes, instead of using program clause resolution to proceed executing, we start consuming answers from the table space. We call these nodes *consumer* nodes. It is interesting to notice that we may call consumer nodes before finding all answers to a producer. Complete execution thus requires complex coroutining between producers and consumers. Prolog engines that support tabling must therefore be able to switch back and forth to consumer nodes, which requires sophisticated mechanisms such as freezing [17] or copying [4].

At consumer node 1 we have no answers stored in the table space. Therefore, the only move we can make is to suspend node 1, and switch back to node 0. We then try the second clause for `path/2`, and obtain a first answer for `path(a,Z)`. Notice that in order to do so we must call `arc(a,Z)`. The `arc/2` procedure is not tabled, hence we say node 2 is an *interior node*. Interior nodes correspond to standard Prolog execution.

Node 2 generates a single answer, which is stored in the table. At this point, we can resume the computation at node 1 with the newly found answer, which in turn leads to a first call to subgoal `path(b,Z)`. The evaluation creates a new tree rooted by `path(b,Z)`, inserts a new entry in the table space for it, and proceeds as for the latter case. The process continues, giving rise to one more tree, for subgoal `path(c,Z)`, and to more answers, one for `path(a,Z)` and the other for `path(b,Z)`.

The example shows the major implementation mechanisms we need to support tabling. In a nutshell:

- Producer nodes require two basic operations: first, we must add new entries and new answers to the table, second, we need to know when a producer is completely evaluated, so that we can close the table. The first operation is called *new answer*, the second *completion*.

- Consumer nodes may be *suspended*, either by freezing the whole stacks [17], or by copying them to separate storage [4]. We must also be able to consume answers from the table and have a mechanism to indicate *which answers* a node has consumed so far. The operation that does this is called *answer resolution*.

- Interior nodes should run just as in standard Prolog.

Arguably, two of the most complex operations in tabling are designing the table itself, and implementing completion. XSB uses tries [12] to implement the table space. Completion is also a difficult problem, as tabled execution may lead to quite intricate dependencies between nodes. XSB uses a completion stack to check at which points completion is possible. Essentially, the completion stack stores the current producer nodes and the dependencies between them. The youngest producer node which does not depend from older producers is called a leader node. Leader nodes define completion points.

3. Or-Parallelism within Tabling

Intuitively, a way to exploit parallelism in tabled programs is by running the producer goals in parallel. This solution has been proposed as Table-Parallelism [5]. Our work was motivated by the observation that exploiting parallelism only at producer nodes may lose substantial parallelism: every node should be a candidate for parallelism [14].

More specifically, in this work we use the OPT model [14]. In this model, each processor (worker) runs most of the time as in sequential tabling. On the other hand, workers without alternatives can steal work from any other worker. We say that in the OPT model parallelism lies above tabling. We chose this model precisely because this separation results in a more structured design, hence simplifying the implementation process.

The question now is whether we can achieve an implementation of the OPT model, and whether that implementation is *efficient*. We implemented OPTYap in order to answer this question. In OPTYap, or-parallelism (ORP) is implemented through copying of stacks. In other words, workers effectively communicate by swapping their whole execution environments. More precisely, we optimize copying by using *incremental copying*, where workers only copy the differences between their stacks. Tabling is implemented by freezing the whole stacks when a consumer blocks. In other words, we cannot recover space above a consumer until *completion*.

OPT requires changes to both the initial designs for parallelism and tabling. First, several workers may be accessing the table simultaneously. Second, producers and consumers may have been set by different workers, in separate stacks. Thus, completion will be distributed between workers. Moreover, we need to know which nodes have real alternatives: there is little point in trying a consumer if the corresponding producer has not produced further answers. We next discuss the major issues in the implementation of OPTYap (please refer to [13] for a detailed presentation of the algorithms discussed here).

3.1. Producers and Consumers

The first major issue in our implementation is precisely how to represent nodes. Whereas in XSB Prolog we had a single stack, in OPTYap we have a separate stack per worker. Our first step was therefore to design three *shared* data structures:

- Nodes that are available for parallel execution are represented by *shared frames*, which store scheduling data.

- Producer nodes are represented in the table space by *subgoal frames*. Each subgoal frame corresponds to a different subgoal call and delimits the *trie structure* representing the answers for the subgoal.

- Consumer nodes are represented by *dependency frames*. Dependency frames are linked in chronological order and they provide concurrent access to the answers found for the corresponding subgoal.

Figure 2 illustrates how producer and consumer nodes interact with the table and dependency spaces. The dependency frames are linked together to form a dependency list of consumer nodes. Additionally, they store information to efficiently check for completion points, and to efficiently move across the dependency graph. This functionality replaces the need for a completion stack.

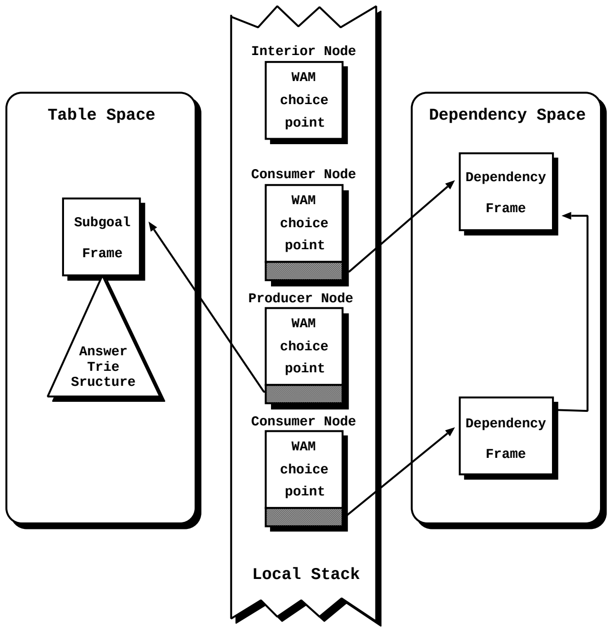


Figure 2. The nodes and their relationship with the table and dependency spaces.

3.2. Completion

Completion is maybe the most difficult problem in tabling. The completion operation is always executed at leader nodes. The operation proceeds by checking whether all descendent consumer nodes have consumed all their answers. Note that in sequential tabling systems, such as in XSB, leader nodes are always producer nodes. This happens because producer nodes are always the youngest points where we can detect that the current subcomputation does not depend from branches above. The story changes in the parallel setting, as Figure 3 shows.

In this example, worker $\mathcal{W}_1$ takes the leftmost alternative while worker $\mathcal{W}_2$ takes the rightmost. While exploiting their alternatives, $\mathcal{W}_1$ calls a tabled subgoal a and $\mathcal{W}_2$ calls a tabled subgoal b. As this is the first call to both subgoals, a producer node is stored for each one. Next, each worker

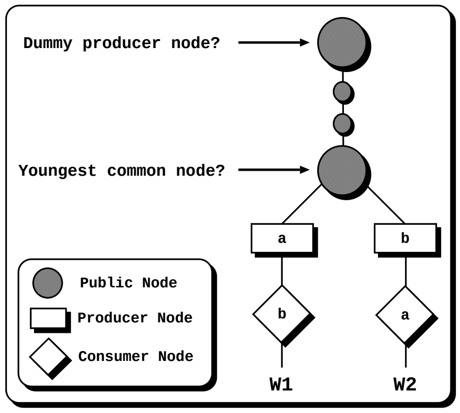


Figure 3. Which is the leader node?

calls the tabled subgoal firstly called by the other, and consumer nodes are therefore allocated. At that point, we may question at which node should we check for completion? Intuitively, we would like to choose a node that is common to both branches and the youngest common node seems the better choice. But that node is an interior node!

We could avoid the problem by disallowing consumer nodes for producer nodes on other worker trees. Unfortunately, such a solution would severely restrict parallelism. Our solution was therefore to allow completion at *public nodes*, that is, at nodes that are shared between nodes.

Waiting for Completion The use of copying for ORP introduces a further complication to completion. Consider the case where worker $\mathcal{W}$ executes several consumer nodes, and then backtracks to a public leader node shared by other workers. Clearly, work is going on below the leader, and $\mathcal{W}$ cannot complete. On the other hand, $\mathcal{W}$ has tried all available alternatives so we would like for $\mathcal{W}$ to try other work. We would like for $\mathcal{W}$ to move anywhere in the tree, say to node $\mathcal{N}$. According to the copying model we use for ORP we should backtrack to the lowest node common to $\mathcal{N}$'s branch, that is, we should reset our stacks to the values of the common node. According to the freezing model that we use for tabling, we cannot recover the current consumers because they are frozen. We thus have a contradiction.

One solution would be to disallow movement in this case. Unfortunately, we would again severely restrict parallelism. Hence, our solution was to copy the consumer goals to an extra space, and store a pointer to the copy from the current leader's shared frame. These suspended computations are considered again when the remaining workers do completion.

Note that this is the only case where ORP and tabling conflict, as this is the only case where we need to move in the tree above the leader node. The reason is that we only recover frozen space when we complete the leader node.

Completion We are now ready to present our parallel completion algorithm. Knowing that we are at the current leader node, the algorithm is actually quite straightforward:

1. Check if we are at a leader node.
2. *Atomically* check whether we are the last worker, store the result in `last_worker`.
3. Check if a consumer node below has unconsumed answers, if so, move to this work even if it was copied away;
4. If `last_worker` is false, try to move above by copying away the current stacks.
5. We have completed.

The synchronization corresponds to checking whether we are the last worker. If we are we can complete. Note that we must take care to check whether we are last before we check for uncompleted answers, as new answers or nodes might have been generated meanwhile.

3.3. The Table

A further problem we had to address in OPTYap was concurrent access to the table. In a nutshell, we can say that there are two critical issues that determines the efficiency of a locking scheme for the table. One is the *lock duration*, that is, the amount of time a data structure is locked. The other is the *lock grain*, that is, the amount of data structures that are protected through a single lock request. It is the balance between lock duration and lock grain that compromises the efficiency of different table locking approaches. For instance, if the lock scheme is short duration or fine grained, then inserting many trie nodes in sequence, corresponding to a long trie path, may result in a large number of lock requests. On the other hand, if the lock scheme is long duration or coarse grain, then going through a trie path without extending or updating its trie structure, may unnecessarily lock data and prevent possible concurrent access by others.

Unfortunately, it was impossible beforehand to know which locking scheme would be optimal. Therefore, OPTYap implements four alternative locking schemes to deal with concurrent accesses to the table space data structures, the *Table Lock at Entry Level* scheme, TLEL, the *Table Lock at Node Level* scheme, TLNL, the *Table Lock at Write Level* scheme, TLWL, and the *Table Lock at Write Level - Allocate Before Check* scheme, TLWL-ABC.

The TLEL scheme essentially allows a single writer per subgoal trie structure and a single writer per answer trie structure. The main drawback of TLEL is the contention resulting from its lock duration scheme. The TLNL enables a single writer per chain of sibling nodes that represent alternative paths from a common parent node. the TLWL scheme is similar to TLNL in that it enables a single writer per chain of sibling nodes that represent alternative paths to a common parent node. However, in TLWL, the common parent node is only locked when writing to the table is likely. TLWL also avoids the TLNL memory usage problem by replacing trie node lock fields with a global array of lock entries. Last, the TLWL-ABC scheme scheme anticipates the allocation and initialization of nodes that are likely to be inserted in the table space to before locking.

3.4. Scheduling

A worker enters in scheduling mode when it runs out of work and only returns to execution whenever a new piece of unexploited work is assigned to it by the scheduler. The scheduler must efficiently distribute the available work for exploitation between workers. In OPTYap, we have the extra constraint of keeping the correctness of sequential tabling semantics.

The OPTYap scheduler engine is mainly based on the YapOr's [15] scheduler algorithm: *when a worker runs out of work it searches for the nearest unexploited alternative in its branch. If there is no such alternative, it selects a busy worker with excess of work load to share work with. If there is no such a worker, the idle worker tries to move to a better position in the search tree*. However, some extensions were introduced in order to preserve the correctness of tabling semantics and to ensure that a worker never moves above a leader until it has fully exploited all alternatives.

4. Performance Evaluation

The main question we wanted to address in our work was whether parallel tabling was worthwhile. Initial results were quite promising [16]. Our original results showed that the overheads for OPTYap over Yap, a fast sequential Prolog system [3], were manageable, in the order of 10% to 20%. We also obtained good initial speedups, but unfortunately only for limited configurations. Our goal in this work is to study whether OPTYap is scalable, and for real applications.

To do so, we used as the main tabled benchmark the XMC model checker [11]. This model checker verifies properties written in the alternation-free fragment of the modal μ-calculus [8] for systems specified in XL, an extension of value-passing CCS [10]. We used standard well-known benchmarks: the specification for `sieve`, `leader` election over 5 processes, and `i-protocol` defined for a correct version (fix) with a huge window size (w = 2)[1].

[1]We are thankful to C.R. Ramakrishnan for providing us these benchmarks.

We further used a set of standard tabling benchmarks. They include `samegen`, which solves the same generation problem for a randomly generated 24x24x2 cylinder; `lgrid`, that computes the transitive closure of a 25x25 grid using left recursion; `lgrid/2` which requires half the relations; and `rgrid/2`, which uses right recursion.

To perform our experiments we used *oscar*, a Silicon Graphics Cray Origin2000 parallel computer from the Oxford Supercomputing Centre. *Oscar* consists of 96 MIPS 195 MHz R10000 processors each with 256 Mbytes of main memory (24 Gbytes of total shared memory) and running the IRIX 6.5.12 kernel. We had access to 32 nodes at most.

Table 1 presents interesting characteristics for each tabled application. The first column shows the execution time, in seconds, for the one worker case. In parentheses, it shows the overhead over sequential execution, that is, without support for parallelism. Support for parallel execution introduces, on average, an overhead between 10% to 20%. The last three columns show the number of producers, and the numbers of unique and repeated answers. Notice that 5 of the benchmarks have a single producer. Most answers generated by the `sieve` and `leader` benchmarks are repeated, indicating the system often reads but hardly writes to the table. In contrast, `lgrid` and `lgrid/2` also have a single producer, but find many different answers, suggesting they often write to the same producer node. Last, `samegen` and `rgrid/2` have several producers, indicating table activity may be well divided.

Bench	One Worker Running Time	# of Prod.	New Answers	
			Unique	Repeated
sieve	268.13(1.14)	1	380	1386181
leader	85.56(1.12)	1	1728	574786
iproto	23.68(1.14)	1	134361	385423
samegen	26.00(1.11)	485	23152	65597
lgrid	4.28(1.21)	1	390625	1111775
lgrid/2	69.02(1.16)	1	160000	449520
rgrid/2	7.51(1.20)	626	781250	2223550

Table 1. Benchmarks characteristics.

Through experimentation, we observed that the locking schemes, TLWL and TLWL-ABC, present the best speedup ratios and they are the only schemes showing scalability. Since none of these two schemes clearly outperform the other, we assumed TLWL as the default. The observed slowdown with higher number of workers for TLEL and TLNL schemes is mainly due to their locking of the table space even when writing is not likely. In particular, for repeated answers they pay the cost of performing locking operations without inserting any new trie node. For these schemes the number of potential contention points is proportional to the number of answers found during execution, being they unique or redundant.

Table 2 presents the speedups for OPTYap with 4, 8, 12, 16, 24 and 32 workers. The table is divided in two main blocks: the upper block groups the benchmarks that showed potential for parallel execution, whilst the bottom block groups the benchmarks that do not show any gains when run in parallel. The speedups are relative to the one worker case of Table 1 and they correspond to the best speedup obtained in a set of 3 runs. Speedup measurements during consecutive runs were rather stable.

Bench	Number of Workers					
	4	8	12	16	24	32
sieve	3.99	7.97	11.94	15.87	23.78	31.50
leader	3.98	7.92	11.84	15.78	23.57	31.18
iproto	3.05	5.08	7.70	9.01	8.81	7.21
samegen	3.72	7.27	10.68	13.91	19.77	24.17
lgrid/2	3.63	7.19	10.21	13.53	19.93	24.35
Average	3.67	7.09	10.47	13.62	19.17	23.68
lgrid	0.65	0.68	0.68	0.55	0.46	0.39
rgrid/2	0.94	1.15	0.92	0.72	0.77	0.65
Average	0.80	0.92	0.80	0.64	0.62	0.52

Table 2. OPTYap's speedups.

The results show superb speedups for the XMC *sieve* and the *leader* benchmarks up to 32 workers. These benchmarks reach speedups of 31.5 and 31.18 with 32 workers, an impressive result considering that XMC is a sophisticated application and that it was ported as is! A more detailed analysis showed that these two benchmarks are quite similar, with a single producer and a single consumer node. Tabling is necessary because **(i)** it avoids a loop in left recursion, and **(ii)** many answers were duplicated. The speedups stem from the interior nodes, and it was our decision to exploit both forms of parallelism which made them possible.

The *iproto* XMC benchmark shows a good result up to 16 workers and then it slows down for 24 and 32 workers. The benchmark again has a single producer and a single consumer. The difference is that the producer generates many more answers, resulting in contention on the table. The `samegen` benchmark also exhibits impressive performance. The benchmark exhibits a respectable number of producers and of consumers, but low contention, hence showing that the computations were well distributed. It is interesting to note that the way we schedule consumer goals changes: as we increase parallelism, more consumer nodes start with unfinished producers. This demonstrates that the completion algorithm is working quite well, and namely that is not causing contention. Last, the `lgrid/2` benchmark is also of the one producer, one consumer style, but with much more contention on the table.

On the other hand, the bottom block shows almost no speedups at all. Only for *rgrid/2* with 8 workers we obtain a slight positive speedup of 1.15. The worst case is for *lgrid* with 32 workers, where we are about 2.5 times slower than execution with a single worker. In this case, surprisingly, we observed that for the whole set of benchmarks the workers are busy for more than 95% of the execution time, even for 32 workers. The actual slowdown is therefore not caused because workers became idle and start searching for work, as usually happens with parallel execution of non-tabled programs. Here the problem seems more complex: workers do have available work, but there is a lot of contention to access that work.

Closer analysis suggested that there are two main reasons that constrain speedups. One relates with massive table access to insert and consume answers. As trie structures are a compact data structure, the presence of massive table access increases the number of contention points. The other relates with the sequencing in the order that answers are found. There are answers that can only be found when other answers are also found, and the process of finding such answers cannot be anticipated. This incurs in high overheads related with suspensions and resumptions of suspended leader branches.

5. Conclusions and Future Work

We have presented the main issues and the performance of OPTYap. Our results show that OPTYap can indeed achieve scalable performance for real applications. This is possible because OPTYap exploits parallelism from both tabled and non-tabled nodes. Our best results were obtained on applications that have a limited number of tabled nodes, but high ORP. On the other hand, we have also obtained good speedups on applications with a large number of tabled nodes.

Table access has been the main factor limiting parallel speedups so far. OPTYap implements tables as tries, thus obtaining good indexing and compression. On the other hand, tries are designed to avoid redundancy. To do so, they restrict concurrency, especially when updating. We plan to study whether alternative designs for the table data-structure can obtain scalable speedups even when frequently updating tables.

Our applications do not show the completion algorithm to be a major factor in performance so far. In the future, we plan to study OPTYap over a large range of applications, namely, natural language, database processing, and non-monotonic reasoning. We expect that non-monotonic reasoning applications, for instance, will raise more complex dependencies and further stress the completion algorithm. We are also interested in the implementation of pruning in the parallel environment.

References

[1] W. Chen, M. Kifer, and D. S. Warren. Hilog: A Foundation for Higher-Order Logic Programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[2] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[3] V. S. Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.

[4] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proceedings of Principles of Declarative Programming*, number 1490 in LNCS, pages 21–35. Springer-Verlag, 1998.

[5] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, number 982 in LNCS, pages 115–132. Springer-Verlag, 1995.

[6] The XSB Group. LMC: The Logic-Based Model Checking Project, 2002. Available from `http://www.cs.sunysb.edu/˜lmc`.

[7] The XSB Group. The XSB Logic Programming System, 2002. Available from `http://xsb.sourceforge.net`.

[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[9] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[10] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[11] C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. Venkatakrishnan. XMC: A logic-programming-based verifi cation toolset. In *Proceedings of Computer Aided Verifi cation*, 2000.

[12] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Effi cient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[13] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Computer Science Department, University of Porto, 2001.

[14] R. Rocha, F. Silva, and V. S. Costa. Or-Parallelism within Tabling. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages*, number 1551 in LNCS, pages 137–151. Springer-Verlag, 1999.

[15] R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artifi cial Intelligence*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.

[16] R. Rocha, F. Silva, and V. S. Costa. On a Tabling Engine that Can Exploit Or-Parallelism. In *Proceedings of the 17th International Conference on Logic Programming*, number 2237 in LNCS, pages 43–58. Springer-Verlag, 2001.

[17] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratifi ed Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.