A distributed tabling system

Carlos Viegas Damásio

Abstract

In this paper we present a general distributed tabling architecture for query evaluation in definite logic programs. The storing of tables and computation of answers is separated, and the algorithms presented have low message overhead. We address the problem of detection of completed tables and provide a solution based on classical distributed algorithms without goal dependency propagation.

Keywords: Logic Programming, Distributed Tabling, Completion Detection.

1 Introduction

A major problem in Prolog Programming is that *SLD(NF)* resolution may not terminate even for datalog programs, i.e. for logic programs without occurrences of function symbols. Furthermore, any *PROLOG* programmer knows that it is necessary to add a lot of control, by means of extra-logical features, in order to try to enforce termination and obtain more efficient programs.

In the literature, two solutions to the termination problem can be found: magic sets and its improvements [17, 1, 2], and tabulation (or memoizing) techniques [20, 21, 22, 16]. Magic sets are a bottom-up approach while tabulation is a top-down one. Aside their differences there is a close relationship between both methods [5, 4]. However, and contrary to what was expected, tabulation methods, and in particular the implementation reported in [18], seem to be more efficient [19]. The main advantages of tabulation are:

- Storing of intermediate results for highly (or often called) recursive predicates, i.e. lemmas.
- Termination is guaranteed for programs that obey the bounded-term-size property of [13].

The author is with A.I. Centre, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2825-114 Caparica, Portugal. E-mail: `cd@di.fct.unl.pt`.

Work partially supported by PRAXIS XXI projects TARDE (Tabulation and Revision in a Distributed Environment), MENTAL (Mental Agents Architecture in Logic) and FLAD-NSF project REAP.

An appropriate abstraction of tabulation technique due to Bol [4] is a good substitute for the usual *SLD* query evaluation procedure for definite logic programs because of its better termination properties. For a different approach to tabling in logic programming see [9]. Our distributed architecture is based on his proposal. Tabling was proven practical by David S. Warren and his colleagues of the XSB Group of SUNY at Stony Brook [7, 18].

Tabulation is nowadays well-understood for the case of definite and normal logic programming. In the midnineties, extensions of tabulation methods for the case of normal logic programs appeared in the literature [7, 3]. Both [7, 3] provide a sound and complete query-evaluation procedure for non-floundering normal logic programs under the well-founded semantics. These are now standard features included in the XSB system.

Tabulation techniques are appropriate to distributed and parallel evaluation, as recognized in [11, 12, 14]. An extension and implementation of a distributed SLG-like query answering procedure is given in [14]. However, that proposal is directed to XSB and relies on the controlled propagation of goal dependencies. This paper presents a simple and open architecture for a distributed tabling system, currently limited to definite logic programs. In the next section we describe the architecure, then discuss the termination and completion detection algorithms. Finally, we discuss the current implementation.

2 A distributed tabling architecture

In this section we provide a high level description of our distributed tabling architecture. We assume a distributed model where message passing is used to communicate among components in the system (processes or threads) suitable for LAN operation. Point-to-point messages are delivered in the order sent, and the send action is reliable and nonblocking. Taken that for granted, we now proceed with the description of the architecture.

The basic distributed architecture requires four types of components: goal manager, table manager, table storage clients, and prover clients. Generally speaking, their functions are the following:

- The goal manager interfaces the distributed tabulation system with the outside world;
- The table manager decides the location of the tables among the several table storage clients, guaranteeing uniqueness of the tables;
- The table storage clients keep the answers for given goal calls, avoiding redundant answers. Simultaneously, they manage the delivery of solutions to the appropriate invoking goals;
- The prover clients perform the logical expansion operations on the set of active goals.

We now describe in more detail each of these components with corresponding pseudo-code. In a running system there exists one goal manager, a table manager, several table storage clients and prover clients. Whenever a user desires to execute a goal in the system a query message should be sent to the goal manager. The query message has three arguments: the list of free variables, the conjunction goal to execute and an external query identifier.

Upon reception of the query message, the goal manager creates a new rule which is sent to all prover clients in order to be asserted[1]. This rule has a new unique predicate symbol in the head with the list of variables in the message and as body the goal to execute. Next, the goal manager sends a call message to the table manager which starts the query evaluation process. Whenever an answer is received by the goal manager, the latter simply sends the corresponding variable substitution to the external caller which invoked the initial query.

Table 1: Pseudo-code for the goal manager

```
State Variables:
    i : counter of goal invocations
    goal : an array mapping goal identifiers to query identifiers
    caller : an array containing the process identifiers that queried the system

Initially
    i := 0

On receipt of query(Vars, Body, N) from P
    set goal[i] := N, caller[i] := P
    create a new rule q_i(Vars) ← Body
    send assert(q_i(Vars) ← Body) to table manager
    send call(q_i(Vars), i) to table manager
    i := i + 1

On receipt of answer(q_i(Vars), i)
    send answer(Vars, goal[i]) to caller[i]
```

The table manager decides where a goal should be tabled. If some client knows neither where a goal is being tabled nor where it should be, then the client sends a call message to the table manager. After determining the client that tables the literal, the table manager informs the sender of the call invocation about where the goal is being tabled and redirects the message to the appropriate table storage client (acting like a proxy).

The table storage client keeps the tables for the goals assigned to it by the table manager and the corresponding table solutions' consumers. Several tables can be maintained by the same table storage client. When a new request for tabling a literal arrives, the table client starts by initializing the necessary data structures:

[1]For simplicity of presentation we assume that the prover clients all have in their database the definite logic program being queried.

Table 2: Pseudo-code for the table manager

State Variables:
  $table$: an array keeping the information where a literal is being tabled

Initially
  $table[Lit] := null$, for every Lit

On receipt of $call(Lit, Gid)$ from P
  if $table[Lit] = null$ then
    select a table storage client c to execute Lit
    set $table[Lit] := c$
  send $table(Lit, c)$ to P
  send $call(Lit, Gid)$ to $table[Lit]$ in behalf of P

On receipt of $assert(H \leftarrow Body)$
  send $assert(H \leftarrow Body)$ to every prover client

gets a new table identifier to be associated with the literal, assigns a consumer to the table, and finally initializes the set of solutions to empty. Subsequently, it sends a call message to a prover client (a prover client can be statically associated with a table storage client or it can be dynamically assigned). If the table client receives a call for a goal which is already being tabled, then it adds the consumer to the set of consumers for the table and sends all the answers already in store. When an answer is received, the table client checks if the solution is "new"[2]. If so, it is stored in the table and sent to all pending consumers.

The prover clients perform the logical operations on goals (see Table 4). A call message deploys an "extension by clause" operation. The prover resolves the literal with the rules matching it in the program and then selects a literal in each resulting resolvent. Each goal has an unique identifier and attached to it is kept, additionally, its head, its body, the selected literal, the storage client, and the corresponding table identifier which initially requested the goal. On reception of a matching answer for a given goal, the prover client unifies the answer with the selected literal and then selects again a new literal in the resulting resolvent. The main operation of the prover client is literal selection[3]. If the body is empty, a solution has been found and it is communicated to the table storage client which requested the computation. Otherwise, a literal is selected in the resolvent and a new goal is activated. The prover then sends a call message to the appropriate table client (or table manager) and waits for answers.

Contrary to the work in [14], we opted for the separation of the table storage clients and prover clients. This results in greater generality but at the cost of more message passing overhead. In our architecture, the construction of solutions for

[2]This may mean either that the received answer is not a variant of a previously found answer or that is not subsumed by a solution.

[3]We assume that the argument variables in $Head$ and $Body$ are renamed each time `select` is called.

Table 3: Pseudo-code for the table storage client

State Variables:
  t : a counter for generating table identifiers
  tabling : an array associating literals with the corresponding table identifiers
  consumer : a array keeping the process and goal identifiers of table consumers
  solution : an array of sets where table answers are stored

Initially
  $t := 0$, $tabling[Lit] := null$ for every Lit

On receipt of $call(Lit, Gid)$ from P
  if $tabling[Lit] = null$ then
    set $Tid := t$, $t := t + 1$
    set $tabling[Lit] := Tid$
    set $consumer[Tid] := \{(P, Gid)\}$
    set $solution[Tid] := \{\}$
    send $call(Lit, Tid)$ to a prover client
  else
    let $Tid := tabling[Lit]$
    set $consumer[Tid] \cup= \{(P, Gid)\}$
    for every Lit in $solution[Tid]$
      send $answer(Lit, Gid)$ to P

On receipt of $answer(Lit, Tid)$
  if Lit not in $solution[Tid]$ then
    $solution[Tid] := solution[Tid] \cup \{Lit\}$
    for every (P, Gid) in $consumer[Tid]$
      send $answer(Lit, Gid)$ to P

a table by more than one prover is not ruled out (without changing the message protocol). At run time, the table storage client may decide to execute the goal in several prover clients (for instance, with distinct computation rules) or it may even split the execution of the table among the prover clients (for instance, some rules are executed in one prover client and the rest in another). Also, the prover client might rebalance its load and send some of its active goals to another client[4], without the intervention or knowledge of the table storage client. These decisions are orthogonal to the proposed architecture.

Furthermore, the table manager may decide to replicate a table to improve response time, for instance. It needs only to send the messages to the desired table storage clients. Also, the attentive reader will notice that the table manager is not strictly necessary in our system. If the goal manager and all prover clients have the knowledge where a literal will be tabled (or can decide on it at run time) then the table manager can be simply removed, obtaining a more decentralized distributed

[4]This requires a new type of message to be handled by the prover clients

Table 4: Pseudo-code for the prover client

```
Initially
    g := 0, for every Lit set table[Lit] := table manager

On receipt of call(Lit, Tid) from P
    for every rule H ← Body such that
    θ is a m.g.u. of H and Lit
        execute select(Hθ, Bodyθ, P, Tid)

On receipt of answer(Lit, Gid) from P
    let θ be a m.g.u. of Lit and selected[Gid]
    execute select(head[Gid]θ, body[Gid]θ, store[Gid], tid[Gid])

On receipt of table(Lit, c)
    set table[Lit] = c

On receipt of assert(H ← Body)
    assert H ← Body to the database

procedure select(Head, Body, P, Tid)
    if Body is empty then
        send answer(Head, Tid) to P
    else
        set Gid := g, g := g + 1
        let Lit a literal in Body
        set selected[Gid] := Lit
        set head[Gid] := Head, body[Gid] := Body − Lit
        set store[Gid] := P, tid[Gid] := Tid
        send call(Lit, Gid) to table[Lit]
```

tabling system. This can be specified in the program through declarations and/or in the rules themselves. The syntax of logic programs can be trivially extended with annotations stating where a literal should be executed (or the rule, if the annotation appears in the head literal).

The message complexity of the system is relatively low. Assuming that the intervention of the table manager is not required[5], calling a goal from a prover client requires 1 or 2 messages. Two messages are transmitted when the table storage client did not already started to table the goal, otherwise 1 is enough. Reporting an answer needs at least 1 message from the prover to the table storage client. If there are n consumers for the table then n additional messages are sent. Thus, an answer requires 1 or $n+1$ messages. The message complexity can be reduced by glueing together a table storage client and a prover associated with that table client. The call message complexity is then at most 1 and the answer communication is reduced

[5] Otherwise, two extra messages are sent.

at least by 1 (more if some of the consumers are local to the same table/prover pair). This new component should comply with the table storage client protocol. Thus, existing tabling systems can be easily included in our architecture (for instance, XSB).

3 Termination and Completion

The architecture described in the previous section does not define how termination of the computation is detected. By the very construction of the protocol of our distributed architecture, it is easily seen that computation ceases when there are no messages in transit in the system and all processes are quiescent. Thus, we can resort to any of the classical distributed termination algorithms, for instance [10, 15], to detect conclusion of the query evaluation process. We also know from the literature that the worst case lower bound complexity for the distributed termination problem is one control message for each basic message [6], and the overhead is relatively small.

The disadvantage of the above strategy is that termination is detected globally. Therefore, the space and structures used for evaluation of concluded subgoals cannot be reclaimed before the whole computation is finished. The problem is of utmost importance when negation by failure is included in the language since negation can only be evaluated after completion of the corresponding subgoal.

A major difficulty (and feature) in the implementation of tabling algorithms is the table completion detection. The XSB system includes in its engine the dynamic construction of strongly connected components (SCCs) of the dependency call graph to perform the completion operation. In [14] is proposed an algorithm and an implementation of a distributed tabling system with completion detection extending XSB. However, the algorithms need to propagate the dependencies among subgoals in the same SCC. We now present an extension of a classical distributed termination algorithm to perform distributed completion of tables without goal dependency propagation.

We resort to the credit recovery termination detection algorithm [15] sketched in Table 5. In this algorithm every process has a weight, which is is greater than zero when the process is active and zero if it is passive. The weight of process p is maintained in variable W_p. There is a central controller which keeps an account of the total of weights of all processes in a state variable, say RC. Initially, RC is equal to the sum of weights of active processes. When a message is sent from process p to process q the weight of p is halved and the other half is sent to q along with the message. If a message arrives to a process then the weight of the message is added to the weight of the process. Whenever a process becomes quiescent it sends to the central controller its current weight via a special decrement message, sets the weight to zero, and enters the passive state. The central controller updates the RC variable upon reception of a decrement message. If RC is zero then all processes have terminated.

Our approach is to use an instance of the credit recovery algorithm for each

Table 5: The Credit Recovery termination detection algorithm

Action	**Effect**
Send_p:	$\{state_p = active\}$ send a message $< M, W_p/2 >$ to q; $W_p := W_p/2$.
Receive_p:	{ A basic message has arrived } receive message $< M, W >$; $state_p := active$ $W_p := W_p + W$
Quiescence_p:	$\{state_p = active\}$ $state_p = passive$; send $< dec\ W_p >$ to controller C; $W_p := 0$
Recover_C:	{A $< dec\ W >$ message has arrived} receive $< dec\ W >$; $RC := RC - W$; **if** $RC = 0$ **then** send $< term >$ to all processes.

strongly connected component of the static dependency graph. A first important remark is that the strongly connected components of the static dependency graph form a directed acyclic graph. This static dependency graph is a safe approximation of the dynamic dependency graph. To each SCC we associate at compile (or even run) time a leader controller client, responsible for starting and completing the SCC termination detection algorithm. The language can be easily extended to encompass declarations of groups of predicates defining a SCC, giving complete freedom to the user. This feature can be complemented with compile time analysis of the program to execute.

In first place, all messages will be labeled with a weight and a zone identifier (or SCC identifier). We assume that the zone identifier 0 is reserved, and will be used to label queries from external processes. The controller for zone 0 is the goal manager. All query messages should be labeled with zone identifier 0. The weight W should be a positive number. Upon reception of a query message the weight is added to RC_0. The processing of the message proceeds as usual, including the steps of the receive action described in Table 5.

A first approach to the completion detection assumes that the leader controller client is a table storage client. Furthermore, a given table storage client keeps all the tables (and only these) for a given SCC. In this case, the changes to the distributed tabling algorithm are minimal and are detailed below.

When a message $call(Lit, Gid)$ is received from zone Z with weight W we check in first place if Lit is in zone Z (the SCC identifier). If not, a call to a literal in other SCC has been issued and the zone Z and weight W are kept in array *consumer*, besides P and Gid. Additionaly, W is summed to $RC_{Z'}$, where Z' is the zone of Lit.

If the table storage client needs to send an answer to other SCC, it consults the consumer table to obtain the zone and current weight availabe. The answer is sent to that zone with halved weight. The other half is put in the consumer's table.

When the storage client concludes that the computation of its SCC has ended (the $RC_{Z'}$ variable reached zero) then the remaining weights in the consumer's table are all returned to the appropriate leader table storage client.

Other approaches are possible, but involve extra book keeping or message passing. In particular, one can allow tables from different SCCs to belong to the same table storage client (this can be achieved with a simple extension to the above proposal). In a more general setting, the leader SCC controller can be separated from the table storage client and the tables of a given SCC might be dispersed among several table storage clients. In this case, every call to a different SCC should be routed throgh the appropriate leader client, which requires an extra message. Also, in a naïve implementation, every answer should be returned via the leader table storage client so that credit change can be performed.

4 Implementation

We have implemented the above architecture in the form of Prolog meta-interpreters on top of PVM-Prolog. PVM-Prolog [8] defines an intermediate layer supporting the parallel and distributed execution of Prolog programs. This model also supports internal concurrency to each process, in the form of user-level threads, achieving a high degree of portability and heterogeneity at the architecture and operating system levels.

PVM-Prolog provides adequate support to a very effective and rapid prototype development. The implementation of our distributed architecture has 130 lines of Prolog source code. Additionally, we implemented a top-level client to interact with the system. These components are entirely implemented on top of XSB-PVM Prolog and Nano-PVM Prolog. To simplify programming of the system there is a layer of PROLOG code supporting basic services like name resolution and distributed termination algorithms. Furthermore, a simulator and elementary distributed debugging and tracing tools were also developed. The system works over a high-speed 8 PC cluster.

The implementation effort will be continued with performance tests and extensions to handle negation by failure. This one of the research topics of the nationally funded PRAXIS XXI project TARDE (Tabulation and Revision in a Distributed Environment).

5 Conclusions and Open Issues

We have presented a open distributed tabling architecture for query evaluation of definite logic programs. The architecture is composed of four types of components: goal manager, table manager, table storage client and prover client. The goal manager interfaces the architecture with the outside world. The table manager selects the clients for storing tables. The table storage client keeps the consumers and solutions of tables. Finally, the prover client performs ordinary query evaluation.

The detection of completed tables is a major issue in tabling environments. We present an approach where dependencies among goals are not communicated among the several components of the architecture. Instead, we resort to a static dependency graph known to all components and determined (or declared) at compile time. The completion algorithm is based on a classical distributed termination detection algorithm (the credit recovery algorithm [15]) with a limited impact on performance and in the generality of the architecture. Several variants of the algorithm were pointed out.

We intend to continue this work with performance and message complexity analysis on tabling benchmarks in order to assess the practical benefits of our algorithm with respect to other proposals in the literature. Furthermore, we will extend the architecture for the evaluation of normal and extended logic programs.

Acknowledgements

We thank Luís Moniz Pereira and Rui Marques for the helpful comments.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways of implementing logic programs. In *5th Symposium on Principles of Database Systems*, 1986.

[2] C. Beeri and R. Ramakrishnan. On the power of magic. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1987.

[3] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *ILPS'93*. MIT Press, 1993.

[4] R. Bol and L. Degerstedt. The underlying search for magic templates and tabulation. In *ICLP'93*. MIT Press, 1993.

[5] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. In *Proc. of DOOD89*, 1989.

[6] K. Mani Chandy and Jayadev Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[7] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *Proc. of the Twelfth Symposium on Principles of Database Systems (PODS'93)*, 1993.

[8] J. C. Cunha and R. F. P. Marques. Distributed algorithm development with pvm-prolog. In *5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 211–2158. IEEE Computer Society Press, 1997.

[9] E. de la Clergerie and B. Lang. LPDA: another look at tabulation in logic programming. In *Proc. of the Int. Conf. on Logic Programming'94*, pages 470–486, 1994.

[10] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11:1–4, 1980.

[11] J. Freire, R. Hu, T. Swift, and D. S. Warren. Parallelizing tabled evaluation. Technical report, SUNY at Stony Brook, 1994. Presented in 1994 Post-ILPS workshop on Parallel Logic Programming Systems.

[12] J. Freire, R. Hu, T. Swift, and D. S. Warren. Parallelizing tabled evaluation. In *7th International PLILP Symposium*, pages 115–132. Springer–Verlag, 1995.

[13] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989.

[14] Rui Hu. *Efficient Tabled Evaluation of Normal Logic Programs in a Distributed Environment*. PhD thesis, State University of New York at Stony Brook, 1997.

[15] Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(4):195–2000, 1989.

[16] W. Nejdl. Recursive strategies for answering recursive queries: the RQA/FQI strategy. In *Proc. of 13th Int. Conf. on Very Large Databases*, 1987.

[17] R. Rohmer, R. Lescouer, and J.-M. Kerisit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–285, 1986.

[18] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficent deductive database engine. In *Proc. of SIGMOD 1994 Conf. ACM*, 1994.

[19] T. Swift and D. S. Warren. Analysis of SLG-WAM evaluation of definite programs. In *ILPS94*, pages 219–235, 1994.

[20] H. Tamaki and T. Sato. OLD resolution with tabulation. In Shapiro, editor, *Proc. of the Int. Conf. on Logic Programming'86*, LNCS 225, pages 84–98. Springer–Verlag, 1986.

[21] L. Vieille. Recursive axioms in deductive databases: the query-subquery approach. In *Proc. of 1st Int. Conf. on Expert Database Systems*, pages 179–193, 1986.

[22] L. Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. of the Int. Conf. on Logic Programming'87*, pages 74–103, 1987.