On Applying Or-Parallelism to Tabled Evaluations

Ricardo Rocha Fernando Silva Vítor Santos Costa
{*ricroc,fds,vsc*}*@ncc.up.pt*

Departamento de Ciência de Computadores & LIACC
Universidade do Porto
Rua do Campo Alegre, 823
4150 Porto, Portugal

Summary

- **Related Work:** Table-Parallelism.
- **Or-Parallelism and Tabled Evaluations:** the fundamental issues in supporting or-parallelism for SLG resolution. Two alternative approaches:
  - **Or-Parallelism within Tabling (OPT)**
  - **Tabling within Or-Parallelism (TOP)**
- **Implementing the OPT Approach:** data areas, data structures and the public completion algorithm required for this approach.
- **Conclusions**

Related Work: Table Parallelism

[J. Freire et al.] presented a first proposal on how to exploit implicit parallelism in tabling systems.

Basic idea:

- each tabled subgoal is associated with a new computational thread, called **generator thread**, that will generate answers and copy them into the table;
- threads that call a tabled subgoal will asynchronously consume answers as they are added to the table by the generator thread.

Table-parallelism views the table as a shared data structure through which cooperating agents may synchronize and communicate.

Table Parallelism: Major Problems

- Ignores potential or-parallelism arising within tabled and non-tabled subgoals.
- Difficults scheduling and load balancing:
  - different sizes of tabled subgoals search space;
  - possibly small number of tabled subgoals;
  - very intricate dependencies between tabled subgoals.
- Efficiency of the new completion algorithm is quite ambiguous:
  - it simultaneously involves different generator threads;
  - requires a considerable number of synchronizations.

Or-Parallelism and Tabled Evaluations

An important advantage of Logic Programming is that parallelism can be exploited implicitly:

- Or-Parallelism;
- And-Parallelism.

An interesting observation is that tabling is still about exploiting alternatives for solving goals:

- it should be amenable for parallel execution within traditional or-parallel execution models;
- no need to restrict parallelism to tabled subgoal calls.

Or-Parallelism and Tabled Evaluations

Key idea: exploit maximum parallelism and take maximum advantage of current technology for or-parallel and tabling systems.

Base system: for efficiency reasons we are most interested in integrating the computation models of:

- **Muse/Aurora** (Or-parallel component) and
- **XSB** (Tabling component).

Main problems: synchronization within tabling operations and scheduling strategies.

Two major approaches to the problem: Or-Parallelism within Tabling (OPT) and Tabling within Or-Parallelism (TOP).

Or-Parallelism within Tabling (OPT)

Workers are considered full SLG-WAM engines: they will spend most of their time executing as if they were sequential engines.

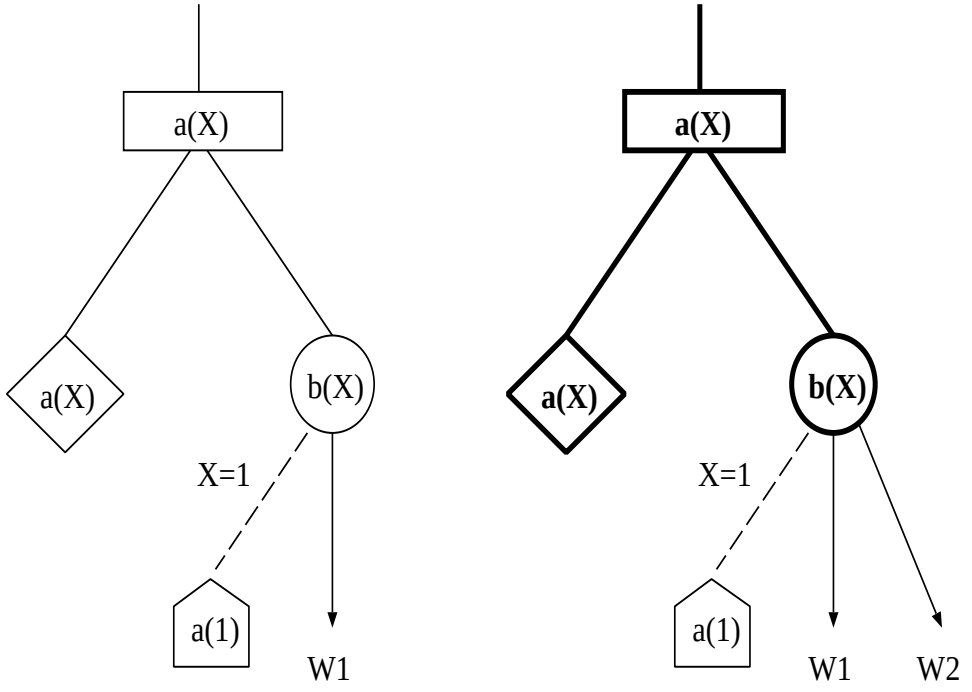
Parallel exploitation: when looking for work, workers take any unexplored alternatives regardless of whether the node it originates from is a generator, active or interior node.

Parallel synchronization: accomplished by extending shared or-frames to include information regarding the SLG resolution. For example, it allows synchronization when workers are executing the **`completion`** or **`answer_return`** operations in the public area.

This approach is close to the environment copy model, as used in the Muse system.

Or-Parallelism within Tabling (OPT): an Example

```
:- table a/1.

a(X) :- a(X).
a(X) :- b(X).

b(1).
b(X) :- ...
b(X) :- ...

?- a(X).
```



One Worker (W1)

Two workers (W1 and W2)

Tabling within Or-Parallelism (TOP)

Workers are considered WAM engines: they only manage a logical branch, not a whole part of the tree.

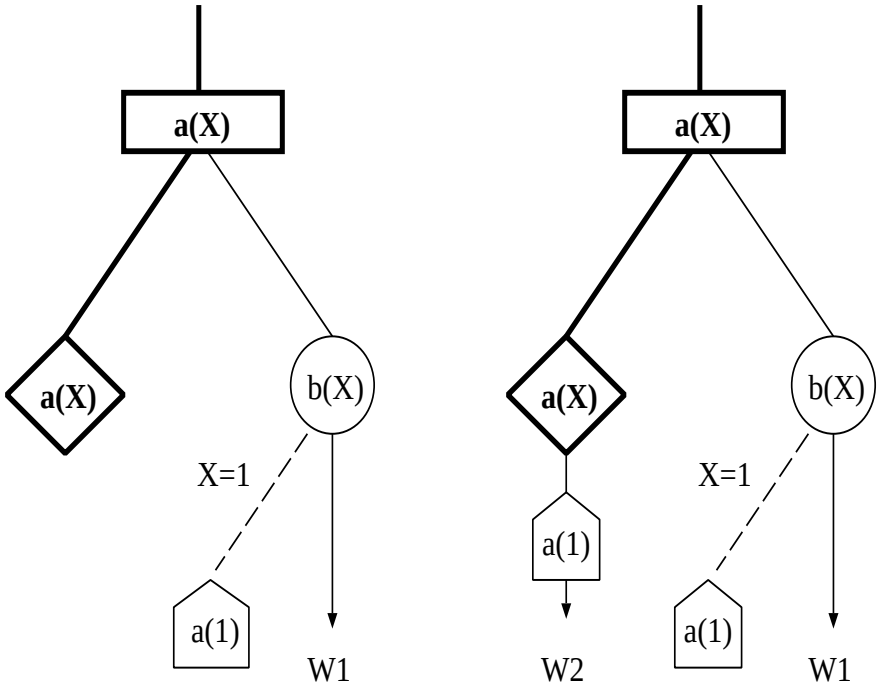
Suspended branches are public branches: when a worker suspends an active node, the node stops being the responsibility of the worker and becomes, instead, shared work that anyone can take. Before leaving the active node, the worker has to make the whole branch public.

The notion of suspension is unified: the system can handle suspensions from or-parallelism and from tabling in the same framework. The unified suspension mechanism must be sufficiently efficient to support all forms of suspension with minimal overhead.

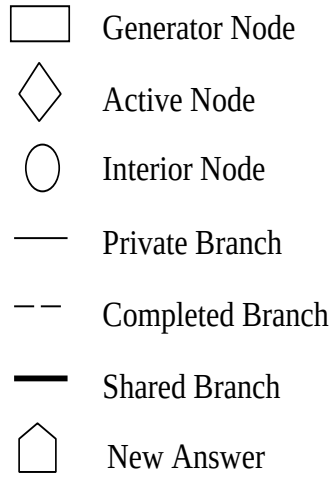
This approach seems closer to the SRI model, as used in the Aurora system.

Tabling within Or-Parallelism (TOP): an Example

```
:- table a/1.

a(X) :- a(X).
a(X) :- b(X).

b(1).
b(X) :- ...
b(X) :- ...

?- a(X).
```



One Worker (W1) Two workers (W1 and W2)



TOP advantages over OPT:

The state of the worker is more clearly defined: a worker occupies the tip of a single branch in the search tree.

Less memory should be spent: a suspended branch will only appear once, instead of possibly several times for several workers.

TOP disadvantages in relation to OPT:

A new SLG engine is required: significant changes to the SLG-WAM engine are required to support the unified suspension.

A suspended branch is a public branch: having a larger public part of the tree may increase overheads.

Suspension must be a very efficient operation: hence this approach is more natural for binding arrays models.

Implementing the OPT Approach

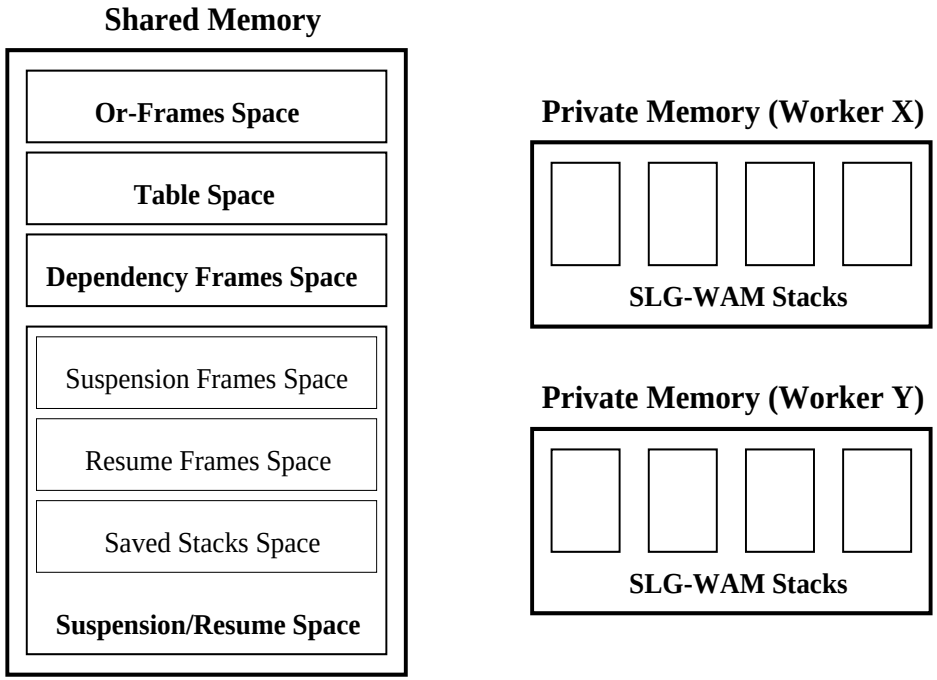
Extend the **YapOr** system to support tabling in a manner close to the **SLG-WAM** engine.

A set of workers will execute a tabled program by traversing its search tree, whose nodes are entry points for parallelism:

- each worker physically owns a copy of the environment (that is SLG-WAM stacks) and shares a large area related to tabling and scheduling;
- a worker with excess of work when prompted for work by other workers, makes public some of their private nodes (using the incremental copy technique);
- whenever a worker backtracks to a public node it synchronizes to perform the operations that SLG-WAM would execute.

Memory Layout

The memory is divided into a large shared area and a number of private areas, corresponding to the number of workers in the system.

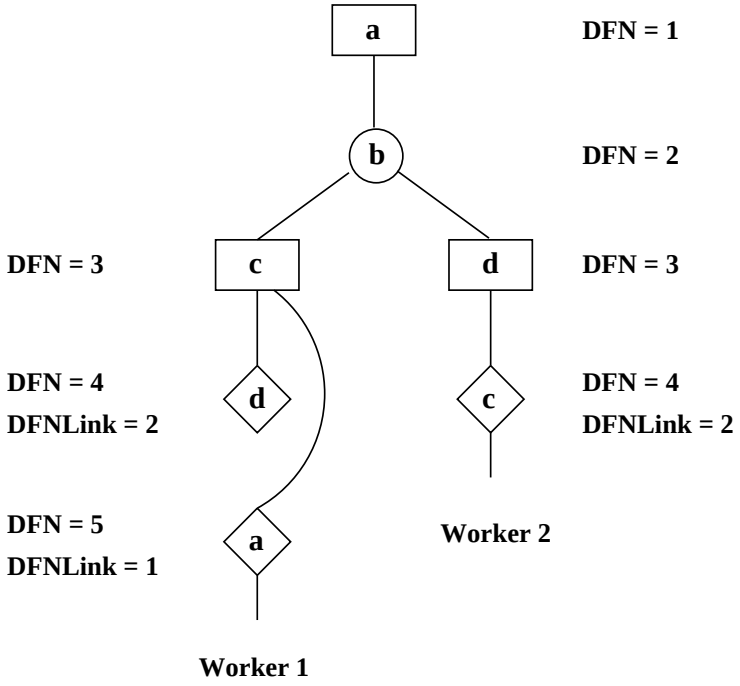


Shared Memory Areas

The shared area is divided into several sub-areas:

- **Or-Frames Space** which is inherited from the or-parallel implementation;
- **Table Space** which is inherited from the sequential tabling implementation;
- **Suspension/Resume Space** which holds information about the suspended branches and about the collected suspended frames to be resumed;
- **Dependency Frames Space**. Each dependency frame holds information about an active node and is shared by the workers that share the corresponding node. Each worker maintains a list of dependency frames to keep track of the active nodes it holds.

Leader Node: DFN and DFNLink Fields



In a worker branch, a node is **leader** when its **DFN** field is equal to the smallest **DFN-Link** field found in the dependency frames of the active nodes below it.

Notice that all kinds of nodes can be leaders in a worker's branch.

Public Completion

The **`public_completion`** instruction implements the algorithm to synchronize the completion operation in the public region and is executed by a worker when it backtracks to a shared node without alternatives or unconsumed answers left.

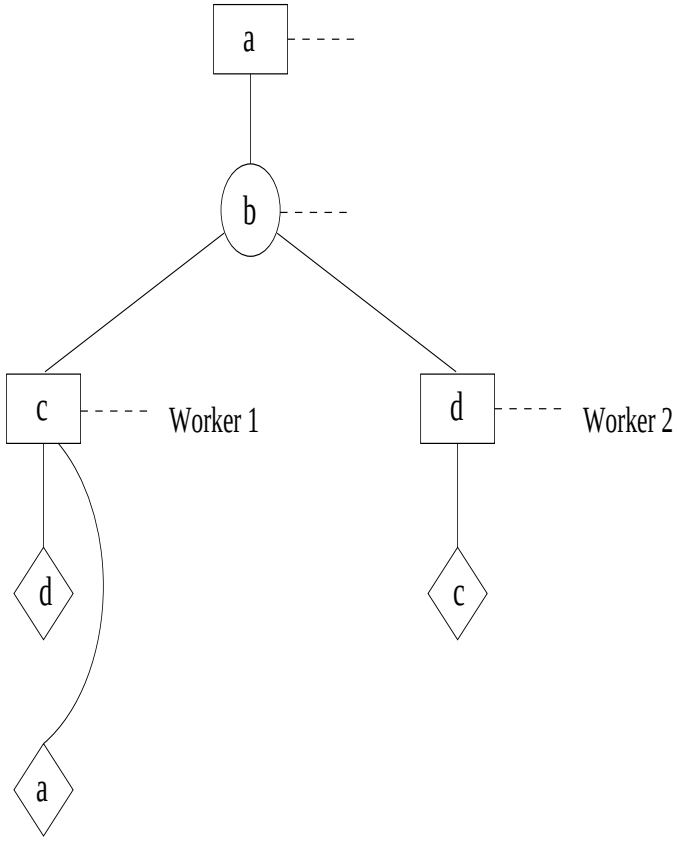
When a leader node is public and contains active nodes below it, this means that it depends on branches explored by other workers.

Thus, after a worker finds a leader node, it may not execute the completion operation immediately. As a result, it becomes necessary to suspend the leader branch.

A worker only completes a leader node when:

- it is the **last worker** in the node;
- there are no **hidden workers** in the node;
- there are no suspended branches **to resume**.

How Public Completion Works



Pseudo-code of the Public Completion Instruction

```
public_completion (node N)

   if (last worker in node)
      for all not collected suspension frames SF stored in node N
         if (exists unconsumed answers for any dependency frame in SF)
            collect (SF) /* to be resumed later */
   if (leader on that node)
      for all dependency frames DF below node N
         if (DF have unconsumed answers)
            backtrack_through_new_answers() /* as in SLG-WAM */
      if (suspension frames collected)
         suspend_current_branch()
         resume (old collected suspension frame)
      else if (N.HiddenWorkers != 0)
         suspend_current_branch()
      else if (last worker in node)
         complete_all()
      else
         suspend_current_branch()
   else   /* not leader */
      if (dependency frames below node N)
         N.HiddenWorkers ++
   backtrack
```

Conclusions

We suggested that there are two major alternatives to tackle the problem of exploiting full or-parallelism in tabling based logic programming systems:

- the Or-Parallelism within Tabling (OPT) approach;
- and the Tabling within Or-Parallelism (TOP) approach.

We presented the fundamental concepts of an environment copying based OPT approach that we believe offers advantages in terms of implementation simplicity and efficiency.

Other then the issues discussed here, support for or-parallelism in tabling systems requires further research in areas such as **scheduling** and **table access**.