Tabling and Or-Parallelism in Yap Prolog: Past, Present and Future

Ricardo Rocha
CRACS & INESC TEC
University of Porto, Portugal
ricroc@dcc.fc.up.pt

Joint work with Fernando Silva, Flávio Cruz, Inês Dutra, João Raimundo, João Santos, Miguel Areias and Vítor Santos Costa

WLPE 2011, Lexington, Kentucky, USA, July 2011
Tabling in Yap: Past

- **Past (2000-2008)**
  - **Tabling Engine [TAPD’00]**
    - Support for sequential tabling based on the SLG-WAM model.
Tabling in Yap: Past

- **Past (2000-2008)**
  - **Tabling Engine [TAPD’00]**
    Support for sequential tabling based on the SLG-WAM model.
  - **Tabling and Implicit Or-Parallelism [ICLP’01, TPLP’05]**
    Support for implicit or-parallelism in tabled logic programs based on the SLG-WAM and environment copying models.
Tabling in Yap: Past

➤ Past (2000-2008)

♦ **Tabling Engine [TAPD’00]**
  Support for sequential tabling based on the SLG-WAM model.

♦ **Tabling and Implicit Or-Parallelism [ICLP’01, TPLP’05]**
  Support for implicit or-parallelism in tabled logic programs based on the SLG-WAM and environment copying models.

♦ **Dynamic Mixed-Strategy Evaluation [ICLP’05]**
  Support for the dynamic intermixing of the two most successful tabling evaluation strategies, batched and local evaluation, at the subgoal level.
Tabling in Yap: Past

**Past (2000-2008)**

- **Tabling Engine [TAPD’00]**
  Support for sequential tabling based on the SLG-WAM model.

- **Tabling and Implicit Or-Parallelism [ICLP’01, TPLP’05]**
  Support for implicit or-parallelism in tabled logic programs based on the SLG-WAM and environment copying models.

- **Dynamic Mixed-Strategy Evaluation [ICLP’05]**
  Support for the dynamic intermixing of the two most successful tabling evaluation strategies, batched and local evaluation, at the subgoal level.

- **Handling Incomplete and Complete Tables [ICLP’06, PADL’07]**
  New techniques for making tabling more efficient when dealing with incomplete tables and more robust when recovering memory from the tables.
Tabling in Yap: Past

Past (2000-2008)

- **Tabling Engine** [TAPD’00]
  Support for sequential tabling based on the SLG-WAM model.

- **Tabling and Implicit Or-Parallelism** [ICLP’01, TPLP’05]
  Support for implicit or-parallelism in tabled logic programs based on the SLG-WAM and environment copying models.

- **Dynamic Mixed-Strategy Evaluation** [ICLP’05]
  Support for the dynamic intermixing of the two most successful tabling evaluation strategies, batched and local evaluation, at the subgoal level.

- **Handling Incomplete and Complete Tables** [ICLP’06, PADL’07]
  New techniques for making tabling more efficient when dealing with incomplete tables and more robust when recovering memory from the tables.

- **Program Transformation with Tabling Primitives** [ICLP’07, PADL’08]
  Support for tabling by applying source level transformations to a tabled program and by using specific external tabling primitives, implemented with the C language interface of Yap, to provide direct control over the search strategy. This work was the basis for tabling support in Ciao Prolog.
Tabling in Yap: Present and Future

Present (2009-2011): already available on Yap’s repository
- Global Trie [PADL’09, ICLP’09, EPIA’11]
- Compact Lists [PADL’10]

Present (2009-2011): to be synchronized soon, hopefully ;)
- Linear Tabling [PADL’10, ICLP’11]
- Call Subsumption [JELIA’10, ICLP’11, EPIA’11]
- Tabling Modes and Answer Subsumption
Tabling in Yap: Present and Future

Present (2009-2011): already available on Yap’s repository
- Global Trie [PADL’09, ICLP’09, EPIA’11]
- Compact Lists [PADL’10]

Present (2009-2011): to be synchronized soon, hopefully ;)
- Linear Tabling [PADL’10, ICLP’11]
- Call Subsumption [JELIA’10, ICLP’11, EPIA’11]
- Tabling Modes and Answer Subsumption

Future (2011-…)
- Multi-Threaded Tabling
- Call Subsumption for Linear Tabling
- Incremental Tabling
- Negation
- Co-Induction
Or-Parallelism in Yap: Past, Present and Future


♦ Or-Parallelism for Shared Memory [EPIA’99, EUROPAR’00]
  Support for implicit or-parallelism based on the environment copying model.

♦ Or-Parallelism For Distributed Memory [EPIA’03]
  Support for implicit or-parallelism based on the stack splitting model.
Or-Parallelism in Yap: Past, Present and Future

  ♦ Or-Parallelism for Shared Memory [EPIA’99, EUROPAR’00]
    Support for implicit or-parallelism based on the environment copying model.
  ♦ Or-Parallelism For Distributed Memory [EPIA’03]
    Support for implicit or-parallelism based on the stack splitting model.

➢ Present (2010-2011)
  ♦ Or-Parallelism using Threads [ICLP’10]
    Redesign of the or-parallel model based on the environment copying model to exploit or-parallelism based on a multi-threaded implementation.
Or-Parallelism in Yap: Past, Present and Future


♦ Or-Parallelism for Shared Memory [EPIA’99,EUROPAR’00]
  Support for implicit or-parallelism based on the environment copying model.

♦ Or-Parallelism For Distributed Memory [EPIA’03]
  Support for implicit or-parallelism based on the stack splitting model.

➢ Present (2010-2011)

♦ Or-Parallelism using Threads [ICLP’10]
  Redesign of the or-parallel model based on the environment copying model to exploit or-parallelism based on a multi-threaded implementation.

➢ Future (2011-…)

♦ Explicit Parallel Constructs
  Use of explicit high-level parallel constructs to trigger parallel execution.

♦ Teams of Workers for Shared/Distributed Memory
  Design of a new parallel platform that combines environment copying with stack splitting to scale-up on clusters of multi-core processors.
In This Talk

➤ Tabling

♦ Global Trie
♦ Call Subsumption
♦ Tabling Modes and Answer Subsumption
♦ Multi-Threaded Tabling

➤ Or-Parallelism

♦ Explicit Parallel Constructs
♦ Teams of Workers for Shared/Distributed Memory
Tabling in Logic Programming

- Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant subcomputations.

- It extends the standard SLD resolution method by adding new tabling operations.
  - First calls to tabled subgoals are evaluated as usual through the execution of Prolog code but answers are inserted into a table space.
  - Similar calls are evaluated by consuming answers from the table space that were generated by the corresponding similar subgoal, instead of re-evaluating them against the program clauses.
  - As new answers are found, they are inserted into the table space and returned to all similar calls.
Table Space

- Can be accessed to:
  - Look up if a subgoal is in the table and, if not, insert it.
  - Look up if a newly found answer is in the table and, if not, insert it.
  - Load answers for similar subgoals.

- Implementation requirements:
  - Fast look-up and insertion methods.
  - Compactness in representation of logic terms.
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
- Each different path through the nodes in the trie corresponds to a term.
- Terms with common prefixes branch off from each other at the first distinguishing token.
Using Tries to Represent the Table Space

Table entry for t/2

VAR0

VAR1

f/1

VAR0

f/1

1

VAR0

subgoal frame for t(VAR0,VAR1)

f/1

2

1

2

1

subgoal frame for t(f(1),VAR0)

f/1

2

1

answer trie

answer trie
GT-T: Global Trie for Terms

- In GT-T, all argument and substitution compound terms appearing in tabled subgoal calls and/or answers are represented only once in the GT, thus preventing situations where these terms are represented more than once in different trie data structures.

- Each path in the original subgoal and answer tries is composed of a fixed number of trie nodes representing the number of argument or substitution terms in the corresponding tabled subgoal call or answer.
GT-T: Global Trie for Terms

Table entry for t/2

VAR0
VAR1

Subgoal frame for t(VAR0, VAR1)

sub1
sub2
sub1
sub2

arg1
VAR0

Subgoal frame for t(f(1), VAR0)

sub1
sub1

Answer trie

f/1
global trie

2 1
GT-ST: Global Trie for Subterms

- The GT-ST maximizes the sharing of the tabled data that is **structurally equal at a second level**, by avoiding the representation of equal compound subterms, and thus preventing situations where the representation of those subterms occur more than once.

- Although GT-ST uses the same GT-T’s tree structure for implementing the GT, every different path in the GT can now **represent a complete term or a subterm of another term, but still being an unique term**.
Consider, for example, the insertion of the terms $f(g(1),g(1))$ and $f(g(2),g(2))$ in the GT-T...

```prolog
:- table t/2.
t(X,Y) :- term(X),
          term(Y).
term(f(g(1),g(1))).
term(f(g(2),g(2))).
```
GT-ST: Global Trie for Subterms

... and in the GT-ST.

\[
\begin{align*}
&\text{:- table t/2.} \\
&t(X,Y) :- \text{term}(X), \text{term}(Y). \\
&\text{term}(f(g(1), g(1))). \\
&\text{term}(f(g(2), g(2))).
\end{align*}
\]
### Global Trie: Experimental Results

<table>
<thead>
<tr>
<th>Terms</th>
<th>Mem</th>
<th>Store</th>
<th>Load</th>
<th>Comp</th>
<th>Mem</th>
<th>Store</th>
<th>Load</th>
<th>Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 ints</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.09</td>
<td>1.11</td>
<td>1.07</td>
</tr>
<tr>
<td>1,000 atoms</td>
<td>1.00</td>
<td>1.04</td>
<td>1.01</td>
<td>1.02</td>
<td>1.00</td>
<td>1.04</td>
<td>1.03</td>
<td>1.08</td>
</tr>
<tr>
<td>1,000 f/1</td>
<td>1.00</td>
<td>1.32</td>
<td>1.16</td>
<td>2.10</td>
<td>1.00</td>
<td>1.34</td>
<td>1.17</td>
<td>2.13</td>
</tr>
<tr>
<td>1,000 f/2</td>
<td>0.50</td>
<td>1.10</td>
<td>1.14</td>
<td>1.84</td>
<td>0.50</td>
<td>1.06</td>
<td>1.11</td>
<td>1.88</td>
</tr>
<tr>
<td>1,000 f/4</td>
<td>0.25</td>
<td>0.81</td>
<td>0.98</td>
<td>1.44</td>
<td>0.25</td>
<td>0.78</td>
<td>1.04</td>
<td>1.53</td>
</tr>
<tr>
<td>1,000 f/6</td>
<td>0.17</td>
<td>0.72</td>
<td>0.72</td>
<td>1.38</td>
<td>0.17</td>
<td>0.66</td>
<td>0.71</td>
<td>1.36</td>
</tr>
<tr>
<td>1,000 [ ]/1</td>
<td>0.50</td>
<td>1.08</td>
<td>1.05</td>
<td>1.61</td>
<td>0.50</td>
<td>1.10</td>
<td>1.02</td>
<td>1.58</td>
</tr>
<tr>
<td>1,000 [ ]/2</td>
<td>0.25</td>
<td>0.80</td>
<td>0.94</td>
<td>1.38</td>
<td>0.25</td>
<td>1.00</td>
<td>1.05</td>
<td>1.48</td>
</tr>
<tr>
<td>1,000 [ ]/4</td>
<td>0.13</td>
<td>0.63</td>
<td>0.54</td>
<td>0.96</td>
<td>0.13</td>
<td>0.89</td>
<td>0.66</td>
<td>1.14</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.53</td>
<td>0.95</td>
<td>0.95</td>
<td>1.42</td>
<td>0.53</td>
<td>0.99</td>
<td>0.99</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Memory usage and store/load times for a t/5 tabled predicate that simply stores in the table space terms defined by `term/1` facts, called with all combinations of one and two free variables in the arguments.
Global Trie: Experimental Results

<table>
<thead>
<tr>
<th>Terms</th>
<th>Mem (MB)</th>
<th>Times (ms)</th>
<th>Mem (MB)</th>
<th>Times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total/GT</td>
<td>Str/Ld/Cmp</td>
<td>Total/GT</td>
<td>Str/Ld/Cmp</td>
</tr>
<tr>
<td>f/1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500,000 g/1</td>
<td>17.17/7.63</td>
<td>126/28/51</td>
<td>1.44 / 2.00</td>
<td>1.55 / 1.14 / 1.00</td>
</tr>
<tr>
<td>500,000 g/3</td>
<td>32.43/22.89</td>
<td>198/34/61</td>
<td>1.24 / 1.33</td>
<td>3.29 / 1.12 / 1.25</td>
</tr>
<tr>
<td>500,000 g/5</td>
<td>47.68/38.15</td>
<td>293/47/83</td>
<td>1.16 / 1.20</td>
<td>1.46 / 1.00 / 0.99</td>
</tr>
<tr>
<td>f/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500,000 g/1</td>
<td>32.43/22.89</td>
<td>203/38/71</td>
<td>1.00 / 1.00</td>
<td>1.28 / 1.13 / 1.09</td>
</tr>
<tr>
<td>500,000 g/3</td>
<td>62.94/53.41</td>
<td>45/60/103</td>
<td>0.76 / 0.71</td>
<td>1.18 / 0.84 / 0.95</td>
</tr>
<tr>
<td>500,000 g/5</td>
<td>93.46/83.92</td>
<td>438/111/146</td>
<td>0.67 / 0.64</td>
<td>1.10 / 0.67 / 0.80</td>
</tr>
<tr>
<td>f/3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500,000 g/1</td>
<td>47.68/38.15</td>
<td>296/50/89</td>
<td>0.84 / 0.80</td>
<td>2.87 / 1.02 / 1.03</td>
</tr>
<tr>
<td>500,000 g/3</td>
<td>93.46/83.92</td>
<td>616/142/164</td>
<td>0.59 / 0.55</td>
<td>1.25 / 0.80 / 0.85</td>
</tr>
<tr>
<td>500,000 g/5</td>
<td>139.24/129.7</td>
<td>832/197/224</td>
<td>0.51 / 0.47</td>
<td>0.96 / 0.67 / 0.74</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>0.96 / 0.97</td>
<td>0.93 / 0.97 / 0.91</td>
</tr>
</tbody>
</table>

Memory usage and store/load times for a t/1 tabled predicate that simply stores in the table space terms defined by term/1 facts.
Call Subsumption

In general, we can distinguish two main approaches to determine similarity between tabled subgoals.

- **Call by Variance:** subgoal \( A \) is similar to \( B \) if they are the same by renaming the variables.
  
  Example: \( p(X,1,Y) \) and \( p(Y,1,Z) \) are **variants** because both can be renamed into \( p(VAR0,1,VAR1) \).
In general, we can distinguish two main approaches to determine similarity between tabled subgoals.

- **Call by Variance:** subgoal $A$ is similar to $B$ if they are the same by renaming the variables.
  Example: $p(X,1,Y)$ and $p(Y,1,Z)$ are **variants** because both can be renamed into $p(\text{VAR0},1,\text{VAR1})$.

- **Call by Subsumption:** subgoal $A$ is similar to $B$ if $A$ is more specific than $B$ (or $B$ is more general than $A$).
  Example: $p(X,1,2)$ is more specific than $p(Y,1,Z)$ because there is a substitution $\{Y=X, Z=2\}$ that makes $p(X,1,2)$ an **instance** of $p(Y,1,Z)$. 
Call Subsumption

► Advantages

♦ Less code is executed because subsumed subgoals can reuse answers instead of executing their own code.
♦ More answers are shared across subgoals, therefore there is less redundancy in the table space.

► Disadvantages

♦ More strict semantics (with some extra-logical features of Prolog, such as the var/1 predicate, call by subsumption should not be used as it can produce wrong results).
♦ The mechanisms to support subsumption-based tabling are harder to implement.
We have developed a new resolution extension called Retroactive Call Subsumption (RCS) that supports subsumption-based tabling by allowing full sharing of answers among subsumptive subgoals, independently of the order they are called.

Example: if \( p(1,X) \) is called before or after \( p(X,Y) \), \( p(1,X) \) will reuse the answers from \( p(X,Y) \). This is not the case in XSB Prolog, because if \( p(1,X) \) is called before \( p(X,Y) \), no reuse will occur.

RCS selectively prunes the evaluation of a subgoal \( S \) when a more general subgoal \( G \) appears later on.

RCS works by pruning the execution branch of \( S \) and then by restarting the evaluation of \( S \) as a consumer. By doing that, we save execution time by not executing code that would generate a subset of the answers we can find by executing \( G \).
Retroactive Call Subsumption: Challenges

- **Keep execution consistent after pruning [JELIA’10]**
  - Build a subgoal dependency tree.
  - Update the low-level stacks related to the pruned subgoals.
  - New operations and evaluation strategies that can handle multiple scenarios in order to ensure correct completion.

- **Compute the set of subsumed subgoals executing [ICLP’11]**
  New algorithms and extensions to the table space to efficiently retrieve the set of subsumed subgoals.

- **Ensure that new consumers will not consume repeated answers [EPIA’11]**
  New table space organization where answers are represented only once.
Retroactive Call Subsumption: Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Yap Prolog Variant/RCS</th>
<th>Yap Prolog Subsumption/RCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>left_first</td>
<td>0.89</td>
<td>0.95</td>
</tr>
<tr>
<td>left_last</td>
<td>0.88</td>
<td>0.90</td>
</tr>
<tr>
<td>double_first</td>
<td>1.07</td>
<td>1.09</td>
</tr>
<tr>
<td>double_last</td>
<td>1.05</td>
<td>1.10</td>
</tr>
<tr>
<td>genome</td>
<td>450.33</td>
<td>0.74</td>
</tr>
<tr>
<td>reach_first</td>
<td>2.54</td>
<td>1.76</td>
</tr>
<tr>
<td>reach_last</td>
<td>3.22</td>
<td>1.87</td>
</tr>
<tr>
<td>flora</td>
<td>3.17</td>
<td>1.17</td>
</tr>
<tr>
<td>fib</td>
<td>1.95</td>
<td>2.02</td>
</tr>
<tr>
<td>big</td>
<td>13.26</td>
<td>13.66</td>
</tr>
</tbody>
</table>

For programs where the time needed to retrieve the answers for the subsumed subgoal offsets the time spent executing the code, RCS performs slightly worse.
Tabling Modes and Answer Subsumption

Mode Declaration

- :- table p(M1,M2,...,Mn).

Available Modes

- index (index argument)
- first (keeps first answer)
- last (keeps last answer)
- all (keeps all answers)
- min (keeps minimum answer)
- max (keeps maximum answer)
### Tabling Modes

```prolog
:- table p(index,index,first).
```

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1,2,10)</td>
<td>New</td>
</tr>
<tr>
<td>p(1,2,6)</td>
<td>Repeats 1</td>
</tr>
<tr>
<td>p(1,3,5)</td>
<td>New</td>
</tr>
<tr>
<td>p(1,3,6)</td>
<td>Repeats 3</td>
</tr>
<tr>
<td>p(1,2,8)</td>
<td>Repeats 1</td>
</tr>
</tbody>
</table>
**Tabling Modes**

```
:- table p(index,index,first).
```

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1,2,10)</td>
<td>New</td>
</tr>
<tr>
<td>p(1,2,6)</td>
<td>Repeats 1</td>
</tr>
<tr>
<td>p(1,3,5)</td>
<td>New</td>
</tr>
<tr>
<td>p(1,3,6)</td>
<td>Repeats 3</td>
</tr>
<tr>
<td>p(1,2,8)</td>
<td>Repeats 1</td>
</tr>
</tbody>
</table>

```
:- table p(index,index,min,all).
```

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1,2,10,[1,3,2])</td>
<td>New + Removed by 2</td>
</tr>
<tr>
<td>p(1,2,6,[1,4,2])</td>
<td>Better Than 1</td>
</tr>
<tr>
<td>p(1,3,5,[1,3])</td>
<td>New</td>
</tr>
<tr>
<td>p(1,3,6,[1,9,3])</td>
<td>Worse Than 3</td>
</tr>
<tr>
<td>p(1,2,6,[1,5,2])</td>
<td>Equal To 2</td>
</tr>
</tbody>
</table>
### Answer Subsumption

:- table p(index,index,min).

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $p(1,2,3)$</td>
<td>New + Removed by 2</td>
</tr>
<tr>
<td>2. $p(1,2,2)$</td>
<td>Better Than 1</td>
</tr>
<tr>
<td>3. $p(2,3,3)$</td>
<td>New</td>
</tr>
<tr>
<td>4. $p(1,Y,1)$</td>
<td>New</td>
</tr>
<tr>
<td>5. $p(1,5,3)$</td>
<td>New</td>
</tr>
</tbody>
</table>
### Answer Subsumption

```prolog
:- table p(index,index,min).
```

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (p(1,2,3)) New + Removed by 2</td>
<td>1</td>
</tr>
<tr>
<td>2. (p(1,2,2)) Better Than 1</td>
<td>2</td>
</tr>
<tr>
<td>3. (p(2,3,3)) New</td>
<td>2 / 3</td>
</tr>
<tr>
<td>4. (p(1,Y,1)) New</td>
<td>2 / 3 / 4</td>
</tr>
<tr>
<td>5. (p(1,5,3)) New</td>
<td>2 / 3 / 4 / 5</td>
</tr>
</tbody>
</table>

```prolog
answer_subsumption(p(_,_,C),min,C).
```

<table>
<thead>
<tr>
<th>Answers</th>
<th>Table Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (p(1,2,3)) New + Removed by 2</td>
<td>1</td>
</tr>
<tr>
<td>2. (p(1,2,2)) Better Than 1+ Removed by 4</td>
<td>2</td>
</tr>
<tr>
<td>3. (p(2,3,3)) New</td>
<td>2 / 3</td>
</tr>
<tr>
<td>4. (p(1,Y,1))</td>
<td>2 / 3 / 4</td>
</tr>
<tr>
<td>5. (p(1,5,3))</td>
<td>2 / 3 / 4 / 5</td>
</tr>
<tr>
<td>4a. (p(1,Y,1))</td>
<td></td>
</tr>
<tr>
<td>4b. (p(1,2,1))</td>
<td>3 / 4a / 4b</td>
</tr>
<tr>
<td>5a. (p(1,5,1))</td>
<td>3 / 4a / 4b / 5a</td>
</tr>
</tbody>
</table>
Multi-Threaded Tabling

- Despite the availability of both threads and tabling in Prolog compilers such as XSB, Yap and Ciao, the implementation of these two features such that they work together is not an easy task.

- Until now, XSB was the only system combining tabling with multi-threading:
  - **Private Tables**: each thread keeps its own copy of the table space, thus avoiding concurrency between threads.
  - **Shared Tables**: when a set of subgoals computed by different threads is mutually dependent, then a usurpation operation synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads.
Multi-Threaded Tabling

- The basis for our work is also on multi-threaded tabling using shared tables, but we propose an alternative view to XSB’s approach.
Multi-Threaded Tabling

- The basis for our work is also on multi-threaded tabling using shared tables, but we propose an alternative view to XSB’s approach.

- In our proposal, each thread was its own tables, i.e., from the thread point of view the tables are private, but at the engine level we use a common table space, i.e., from the implementation point of view the tables are shared among all threads.
Multi-Threaded Tabling

The basis for our work is also on multi-threaded tabling using shared tables, but we propose an alternative view to XSB’s approach.

In our proposal, each thread was its own tables, i.e., from the thread point of view the tables are private, but at the engine level we use a common table space, i.e., from the implementation point of view the tables are shared among all threads.

We propose three designs for our common table space approach:

- **No-Sharing** (similar to XSB with private tables)
- **Subgoal-Sharing**
- **Full-Sharing**
Multi-Threaded Tabling: No-Sharing

Memory usage for a table $T$ assuming $NK$ threads evaluating $NS$ subgoals:

$$\text{sizeof}(\text{TE}) + \text{sizeof}(\text{BA}) + [\text{sizeof}(\text{STS}) + [\text{sizeof}(\text{SF}) + \text{sizeof}(\text{ATS})] * \text{NS }] * \text{NK}$$
Multi-Threaded Tabling: Subgoal-Sharing

Memory usage for a table T assuming NK threads evaluating NS subgoals:

\[
\text{sizeof(TE)} + \text{sizeof(STS)} + \left[ \text{sizeof(BA)} + \left[ \text{sizeof(SF)} + \text{sizeof(ATS)} \right] \right] \times NK \] \times NS
Multi-Threaded Tabling: Full-Sharing

Memory usage for a table \( T \) assuming \( NK \) threads evaluating \( NS \) subgoals:

\[
\text{sizeof}(TE) + \text{sizeof}(STS) + \left[ \text{sizeof}(SE) + \text{sizeof}(BA) + \text{sizeof}(ATS) + \text{sizeof}(SF) \times NK \right] \times NS
\]
## Multi-Threaded Tabling: Experimental Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pyramid 400</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>322,799</td>
<td>370,894,136</td>
</tr>
<tr>
<td>SS</td>
<td>1.12</td>
<td>-448,984</td>
</tr>
<tr>
<td>FS</td>
<td>4.30</td>
<td>-346,599,384</td>
</tr>
<tr>
<td><strong>Cycle 400</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>209,678</td>
<td>247,351,736</td>
</tr>
<tr>
<td>SS</td>
<td>1.13</td>
<td>-225,544</td>
</tr>
<tr>
<td>FS</td>
<td>4.84</td>
<td>-231,333,792</td>
</tr>
<tr>
<td><strong>Grid 20</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>193,419</td>
<td>247,351,736</td>
</tr>
<tr>
<td>SS</td>
<td>0.96</td>
<td>-225,544</td>
</tr>
<tr>
<td>FS</td>
<td>1.14</td>
<td>-231,333,792</td>
</tr>
</tbody>
</table>

Execution time (in milliseconds) and total memory usage (bytes) running 16 simultaneous threads, all executing the same query goal.
### Multi-Threaded Tabling: Experimental Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Execution</th>
<th>Tries</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TE</td>
<td>SF</td>
<td>STS</td>
</tr>
<tr>
<td>Pyramid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>56</td>
<td>921,600</td>
<td>1,024,640</td>
</tr>
<tr>
<td>SS</td>
<td>0</td>
<td>0</td>
<td>-960,600</td>
</tr>
<tr>
<td>FS</td>
<td>0</td>
<td>-307,200</td>
<td>-960,600</td>
</tr>
<tr>
<td>Cycle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>56</td>
<td>461,952</td>
<td>513,920</td>
</tr>
<tr>
<td>SS</td>
<td>0</td>
<td>0</td>
<td>-481,800</td>
</tr>
<tr>
<td>FS</td>
<td>0</td>
<td>-153,984</td>
<td>-481,800</td>
</tr>
<tr>
<td>Grid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NS</td>
<td>56</td>
<td>461,952</td>
<td>513,920</td>
</tr>
<tr>
<td>SS</td>
<td>0</td>
<td>0</td>
<td>-481,800</td>
</tr>
<tr>
<td>FS</td>
<td>0</td>
<td>-153,984</td>
<td>-481,800</td>
</tr>
</tbody>
</table>

Specific memory usage (bytes) running 16 simultaneous threads, all executing the same query goal.
Explicit Parallel Constructs

Traditional parallel LP systems usually run in parallel mode from beginning to end and this may severely restrict parallelism when supporting sequential semantics.

- If we avoid exploiting non-leftmost sub-computations, we may be restricting the granularity of the available parallel work.
- If we allow such sub-computations, we may be executing speculative work and/or side-effects that would not be done in a sequential system.
Explicit Parallel Constructs

- Traditional parallel LP systems usually run in parallel mode from beginning to end and this may severely restrict parallelism when supporting sequential semantics.
  - If we avoid exploiting non-leftmost sub-computations, we may be restricting the granularity of the available parallel work.
  - If we allow such sub-computations, we may be executing speculative work and/or side-effects that would not be done in a sequential system.

- However, most of the execution time is spent in a parallel application is spent in computations that are inherently parallel and independent and only a small part of the execution time is spent in sequential parts of code:
  - Initialization code
  - Code to partitioning the data into small sub-tasks
  - Code to aggregate/reduce data from different sub-tasks
Explicit Parallel Constructs

Most of the recent proposals on parallel programming, where parallelism is exploited explicitly, are trying to **encapsulate** some of the low-level details in more **high-level explicit parallel constructs for well-know patterns** and let the execution model implement them **implicitly**:

- **OpenMP**
- **Intel Threading Building Blocks**
- **Map-Reduce**
Explicit Parallel Constructs

Most of the recent proposals on parallel programming, where parallelism is exploited explicitly, are trying to encapsulate some of the low-level details in more high-level explicit parallel constructs for well-known patterns and let the execution model implement them implicitly:

- OpenMP
- Intel Threading Building Blocks
- Map-Reduce

Our approach goes in the opposite direction. It establishes its foundations on implicit parallelism and relies on high-level explicit parallel constructs to trigger parallel execution.

- More declarative, thus simplifying parallel programming;
- Better performance, since we can benefit from the intrinsic and strong potential that LP has for implicit parallelism;
- More general, can be easily generalized to implement new parallel constructs with minor changes to the low-level parallel engine.
Explicit Parallel Constructs

Some basic parallel constructs we are interested in are:

- `parallel/1`
- `parallel_findall/3`
- `parallel_once/1`
Explicit Parallel Constructs

Some basic parallel constructs we are interested in are:

- `parallel/1`
- `parallel_findall/3`
- `parallel_once/1`

```prolog
go1 :- statistics(cputime,[Init,_]),
    parallel(benchmark),
    statistics(cputime,[End,_]),
    Time is End - Init, writeln(Time).
```
Explicit Parallel Constructs

Some basic parallel constructs we are interested in are:

- `parallel/1`
- `parallel_findall/3`
- `parallel_once/1`

```
go1 :- statistics(cputime,[Init,_]),
       parallel(benchmark),
       statistics(cputime,[End,_]),
       Time is End - Init, writeln(Time).

go2 :- init_something,
       parallel_findall(X,benchmark(X),L),
       do_something_with_results(L).
```
Explicit Parallel Constructs

As in OpenMP, we can extend the parallel constructs to include pre-defined directives that can be used to instruct and/or to pass specific information to the execution system about the computation at hand:

- num_workers(expr)
- execution_model(env_copying | stack_splitting)
- if(expr)
- reduction(var,operator)
- cut_safe
- allow_out_of_order_side_effects
Explicit Parallel Constructs

As in OpenMP, we can extend the parallel constructs to include **pre-defined directives** that can be used to instruct and/or to pass specific information to the execution system about the computation at hand:

- `num_workers(expr)`
- `execution_model(env_copying | stack_splitting)`
- `if(expr)`
- `reduction(var,operator)`
- `cut_safe`
- `allow_out_of_order_side_effects`

```prolog
go :- init_something(I),
    parallel(benchmark(X), [if(I), reduction(X,sum), cut_safe]),
    do_something_with_result(X).
```
Teams of Workers for Shared/Distributed Memory

In the past, we have already designed and developed or-parallel systems for shared and distributed memory architectures:

- **Shared Memory**: support for implicit or-parallelism based on the *environment copying model*.
- **Distributed Memory**: support for implicit or-parallelism based on the *stack splitting model*.
Teams of Workers for Shared/Distributed Memory

In the past, we have already designed and developed or-parallel systems for shared and distributed memory architectures:

- **Shared Memory**: support for implicit or-parallelism based on the environment copying model.
- **Distributed Memory**: support for implicit or-parallelism based on the stack splitting model.

Design a new parallel platform that will be able to **take advantage of both models to scale-up on clusters of multi-core processors**.

For that, we will consider **Teams of Workers**, i.e., workers sharing the same memory address space. Workers executing in different computer nodes cannot belong to the same team, but we can have more than a team in the same computer node.
Teams of Workers for Shared/Distributed Memory

Workers inside a team (shared memory only) can distribute work using:

- Environment Copying
- Stack Splitting

Teams of workers can distribute work using:

- Environment Copying (shared memory only)
- Stack Splitting (shared and distributed memory)

This idea is similar to the MPI/OpenMP hybrid programming pattern where MPI is usually used to communicate work among workers in different computer nodes and OpenMP is used to communicate work among workers in the same node.

By invoking our explicit parallel constructs with proper directives, we will be able to trigger parallel execution of these different combinations of number of workers, teams of workers and execution models.
Teams of Workers for Shared/Distributed Memory
Teams of Workers for Shared/Distributed Memory

stack_splitting
### Teams of Workers for Shared/Distributed Memory

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### env_copying

- CPU
- CPU
- CPU
- CPU

#### stack_splitting

- CPU
- CPU
- CPU
- CPU

#### Network
Thank You!

Ricardo Rocha
CRACS & INESC TEC
University of Porto, Portugal

ricroc@dcc.fc.up.pt
http://www.dcc.fc.up.pt/~ricroc

Yap Prolog:  http://www.dcc.fc.up.pt/~vsc/Yap
Project LEAP:  http://www.dcc.fc.up.pt/leap