Or-Parallel Prolog Execution on Multicores Based on Stack Splitting

Rui Vieira, Ricardo Rocha and Fernando Silva

CRACS & INESC TEC
Faculty of Sciences, University of Porto

DAMP 2012
Declarative Aspects and Applications of Multicore Programming

Why Parallelism in Prolog?

Efficient sequential implementations

- There are many efficient sequential implementations of Prolog, mostly based on the **Warren Abstract Machine (WAM)**.

Potential for implicit parallelism

- Prolog programs **naturally exhibit implicit parallelism**, thus **freeing** the programmers from the task of **explicitly** identifying it.
- This makes parallel logic programming **as easy as** logic programming.

Implicit Parallelism in Prolog

And-parallelism

- is the simultaneous evaluation of the several Prolog subgoals in the body of a clause.

```
path(X,Z) :- path(X,Y), edge(Y,Z).
```

Implicit Parallelism in Prolog

And-parallelism

- is the simultaneous evaluation of the several Prolog subgoals in the body of a clause.

```
path(X,Z) :- path(X,Y), edge(Y,Z).
```

Or-parallelism

- is the simultaneous evaluation of a Prolog goal against all the alternative predicate clauses that match that goal.

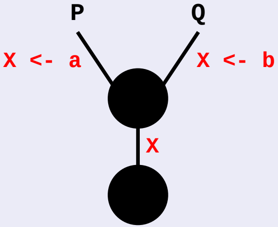
```
path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).
```

- The least complexity of or-parallelism (alternative matching clauses are independent of each other) makes or-parallel models **more attractive** and **more successful** at a first step.

Or-Parallelism: Implementation Challenges

Multiple bindings

- How to efficiently represent the multiple bindings for variables shared by the parallel execution of alternative clauses.



- Private areas to store the bindings for each branch are required:
  - **Binding arrays**
  - **Environment copying**

Or-Parallelism: Implementation Challenges

Scheduling

- Achieving the necessary cooperation, synchronization and concurrent access to shared data structures among several workers during their execution is a difficult task.
- A parallel Prolog system is no exception as the parallelism that Prolog programs exhibit is usually highly irregular:
  - **Topmost dispatching** or **bottommost dispatching**.
  - **Dynamic sharing** or **static sharing** (stack splitting).

Our Goal

Stack splitting on multicores

- Design and implement static sharing, namely **stack splitting**, in the **YapOr system**.
- Benefit from prior research on the development of the YapOr system and extend it to efficiently support two work sharing stack splitting models, namely **vertical splitting** and **half splitting**, on multicore architectures.

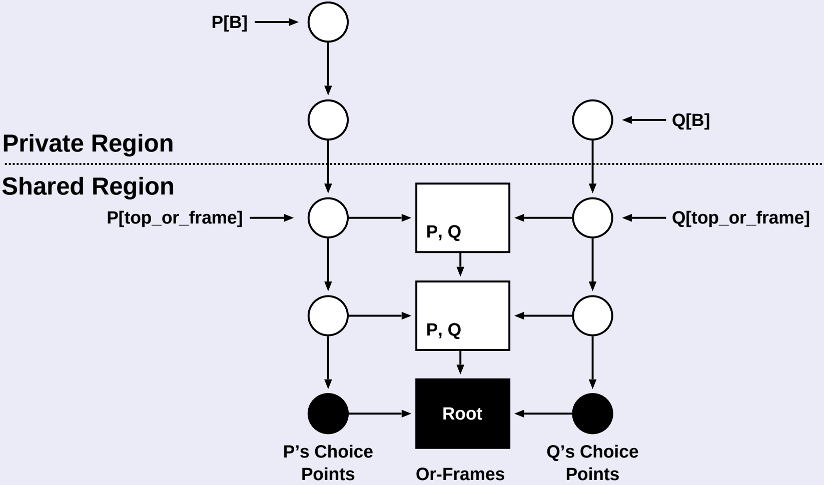
The YapOr System

Execution Model

- YapOr is based on the **environment copying model**:
  - Each worker maintains a **separated copy** of its environment.
  - Sharing is done by **copying the execution stacks** between workers.
- YapOr's scheduler is based on **bottommost dispatching** and **dynamic sharing**.
  - Synchronization is mostly needed at work sharing operations to ensure that each alternative is **explored only once**.
  - Shared nodes are represented by **or-frames**, a data structure that workers must access to obtain the untried alternatives, point in which **mutual exclusion** is enforced.

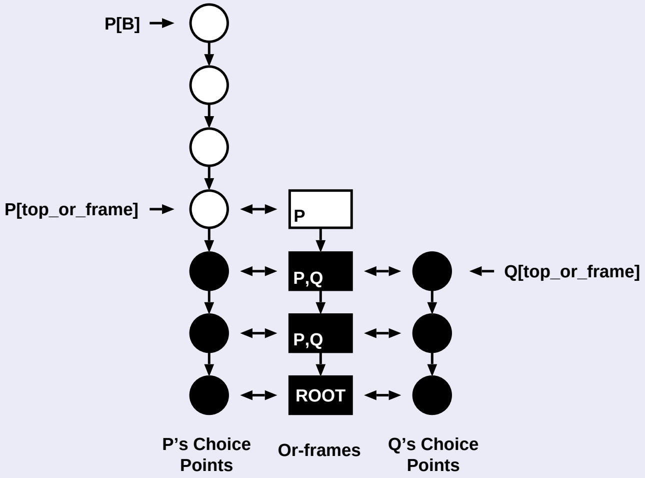
The YapOr System

Data structures



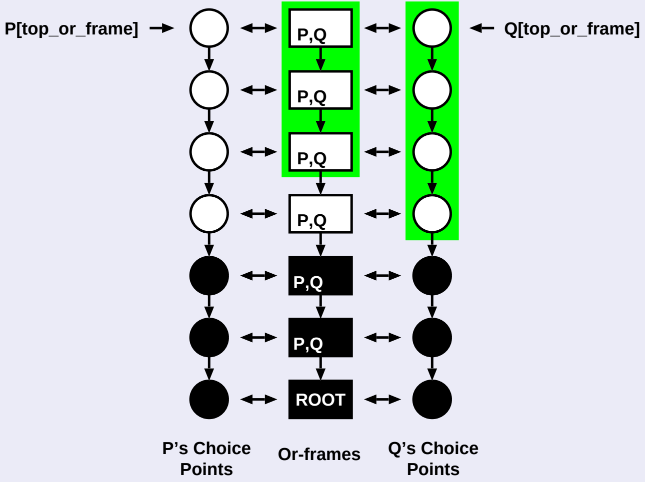
Environment Copying

Before sharing



Environment Copying

After sharing



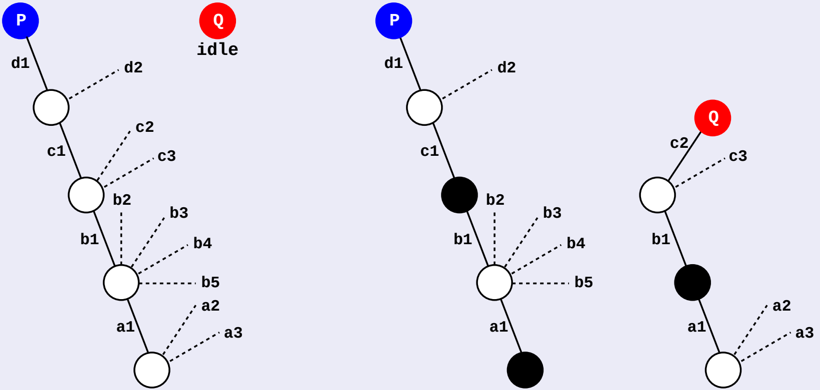
Stack Splitting

General ideas

- Introduced to target **distributed memory architectures**.
- Aiming to **avoid the mutual exclusion requirements** when accessing shared branches of the search tree.
- Defines a work sharing strategy in which all available work is divided **in two fully independent parts**.
- The splitting is such that both workers can continue executing its branch of computation **independently**, without any need for further synchronization.

Stack Splitting

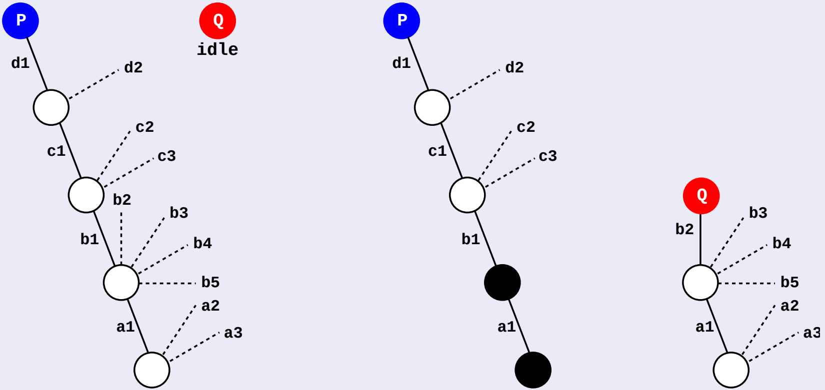
Vertical splitting



(a) before sharing (b) after sharing

Stack Splitting

Half splitting

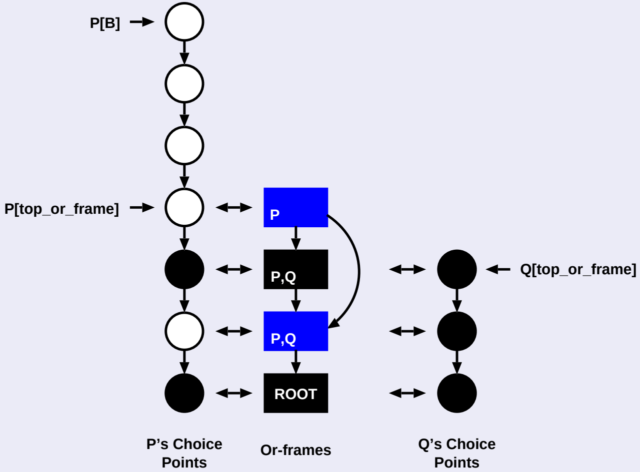


(a) before sharing

(b) after sharing

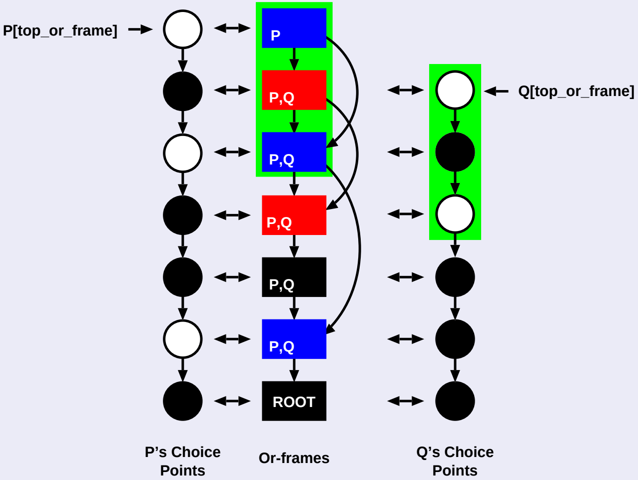
Vertical Splitting

Before sharing



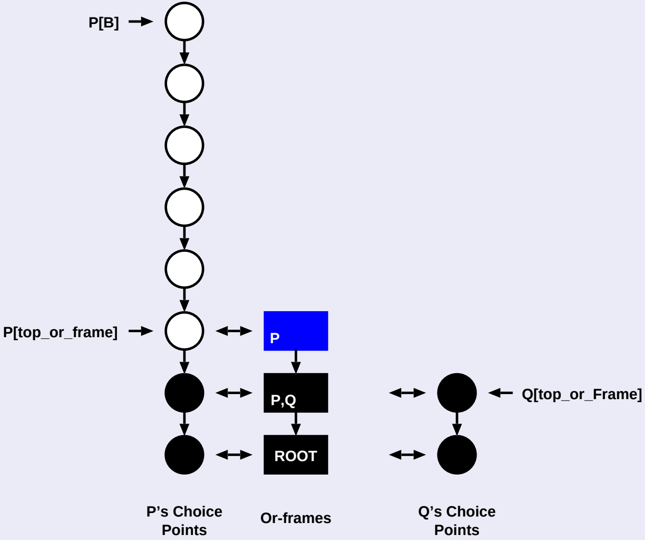
Vertical Splitting

After sharing



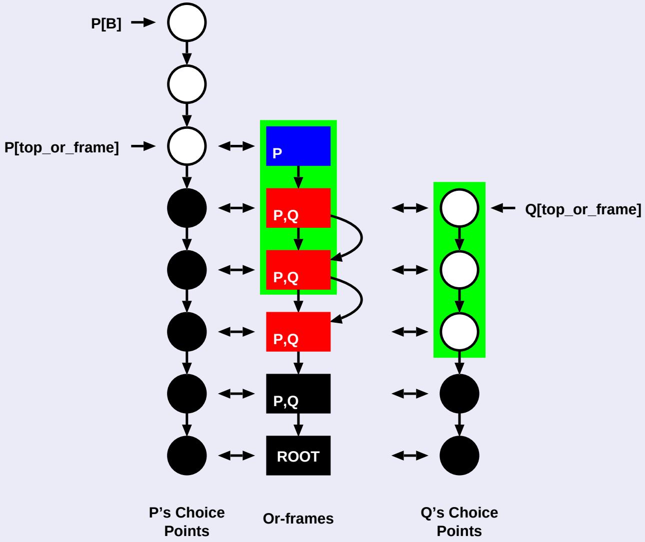
Half Splitting

Before sharing



Half Splitting

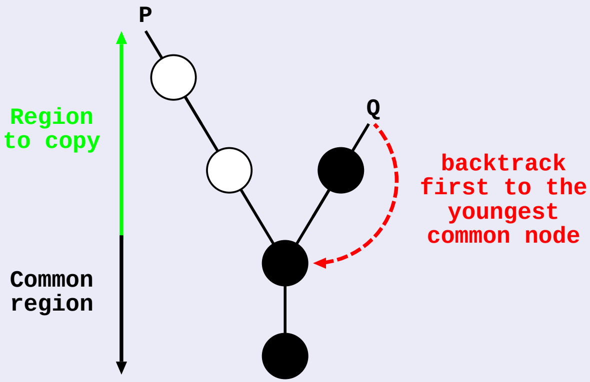
After sharing



Incremental Copy

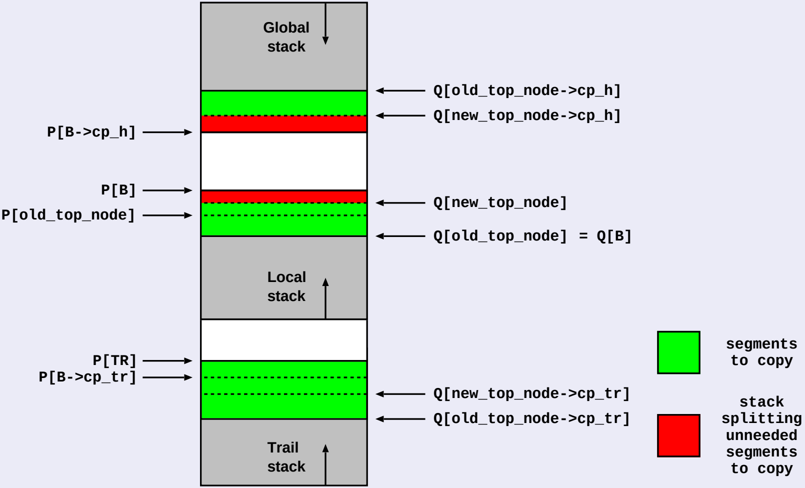
General idea

- Aims to **minimize the amount of data** copied between P and Q.
- It copies only the **state difference** between workers P and Q.



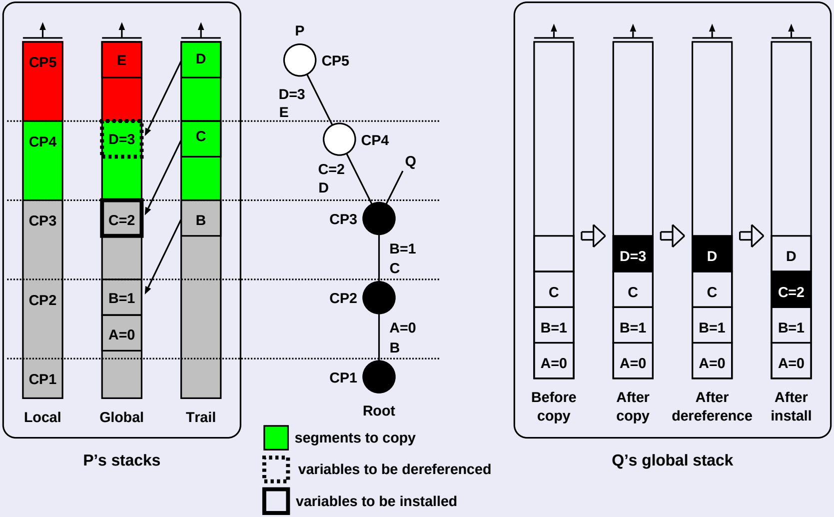
Incremental Copy

Copy ranges



Incremental Copy

Dereference phase



Experimental Results

Environment

- A machine with 4 AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores in total) and 64 (4x16) GB of DDR-2 667MHz RAM.
- Running Linux kernel 2.6.31.5-127 64 bits with Yap Prolog 6.2.0.
- All benchmarks find all the solutions by simulating an automatic failure whenever a new solution is found.
- Each benchmark was executed 20 times and the results are the average of those 20 executions.

Experimental Results

Cost of the parallel models (1 worker)

Benchmarks	Yap (s)	YapOr(1w) / Yap		
		EC	VS+IC	HS+IC
cubes7	0.202	1.044	1.038	1.059
ham	0.321	1.198	1.197	1.098
magic	45.990	0.985	0.986	0.901
map	22.434	1.130	1.130	1.141
nsort10	2.567	1.140	1.149	1.040
nsort11	28.239	1.135	1.133	1.028
nsort12	339.406	1.126	1.129	1.003
puzzle	0.154	1.152	1.151	1.106
puzzle4x4	9.875	1.032	1.030	0.958
queens13	48.220	1.061	1.063	1.001
Average		1.100	1.101	1.033

Experimental Results

Environment copying

Benchmarks	Workers			
	4	8	16	24
cubes7	3.27	**5.66**	**7.62**	**7.43**
ham	3.10	5.34	**7.32**	**6.49**
magic	4.05	8.08	16.08	23.95
map	3.58	7.11	13.92	20.32
nsort10	3.61	7.08	13.44	**17.97**
nsort11	3.71	7.37	14.63	21.63
nsort12	3.68	7.39	14.89	22.19
puzzle	**2.96**	**4.68**	**5.94**	**5.03**
puzzle4x4	3.90	7.77	15.32	22.44
queens13	3.76	7.50	**14.93**	**22.22**
Average	3.56	**6.80**	**12.41**	**16.97**

Experimental Results

Vertical splitting with (without) incremental copy

Benchmarks	Workers			
	4	**8**	**16**	**24**
cubes7	**3.33** (2.63)	5.52 (3.34)	6.98 (3.00)	6.05 (2.41)
ham	3.11 (2.39)	**5.36** (3.21)	7.00 (3.29)	5.00 (2.94)
magic	4.04 (4.04)	8.07 (8.00)	16.04 (15.80)	23.79 (23.11)
map	**3.59** (3.58)	**7.13** (7.05)	**13.96** (13.59)	**20.36** (19.52)
nsort10	3.58 (3.52)	7.00 (6.52)	13.17 (9.81)	17.56 (10.41)
nsort11	3.66 (3.71)	7.26 (7.32)	14.33 (14.09)	21.16 (19.93)
nsort12	3.63 (3.69)	7.27 (7.39)	14.60 (14.85)	21.77 (22.05)
puzzle	2.93 (1.96)	4.52 (1.88)	5.23 (1.58)	4.27 (1.21)
puzzle4x4	3.90 (3.88)	7.76 (7.63)	15.32 (14.62)	22.46 (20.42)
queens13	3.75 (3.73)	7.46 (7.36)	14.77 (14.23)	21.93 (20.54)
Average	3.55 (3.31)	6.74 (5.97)	12.14 (10.49)	16.44 (14.26)

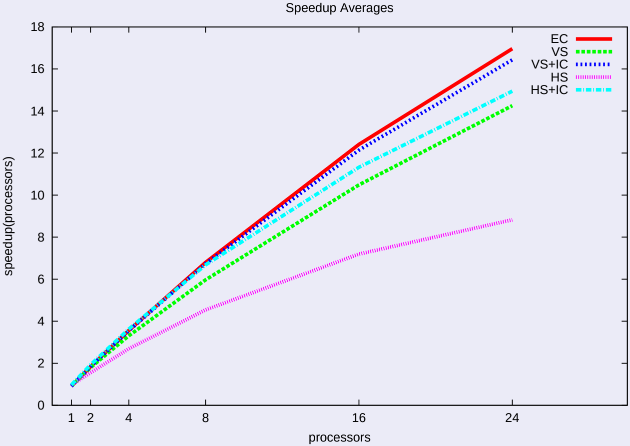
Experimental Results

Half splitting with (without) incremental copy

Benchmarks	Workers			
	4	**8**	**16**	**24**
cubes7	3.03 (0.71)	4.67 (0.77)	5.18 (0.59)	3.65 (0.41)
ham	**3.22** (1.52)	5.22 (1.85)	5.56 (1.90)	4.05 (1.63)
magic	**4.44** (4.15)	**8.80** (7.64)	**17.44** (13.34)	**25.86** (16.45)
map	3.35 (1.72)	5.36 (2.49)	5.89 (2.58)	4.86 (2.29)
nsort10	**3.68** (3.27)	**7.34** (5.76)	**13.49** (8.45)	17.91 (8.95)
nsort11	**3.78** (3.69)	**7.58** (7.19)	**14.90** (13.17)	**22.06** (18.54)
nsort12	**3.79** (3.76)	**7.58** (7.47)	**15.36** (14.68)	**22.76** (21.18)
puzzle	2.96 (1.62)	4.48 (1.77)	5.01 (1.58)	4.46 (1.27)
puzzle4x4	**4.13** (3.83)	**8.08** (6.82)	**15.60** (11.42)	**22.93** (13.59)
queens13	**3.91** (2.66)	**7.69** (3.68)	14.82 (4.24)	20.90 (3.97)
Average	**3.63** (2.69)	6.68 (4.54)	11.33 (7.20)	14.94 (8.83)

Experimental Results

Overall average analysis



Conclusions and Further Work

Conclusions

- We have presented the design and implementation of two stack splitting models in the YapOr system.
- Although stack splitting was proposed for distributed memory, our results show that it is equally suitable for shared memory machines:
  - In many benchmarks, we achieved **speedups above 20 on 24 cores**.
  - Vertical splitting overall performance **close to original YapOr**.
  - Half splitting performed **better in 4 of 10 benchmarks**.
  - Incremental copy **clearly benefits performance**.

Conclusions and Further Work

Further Work

- Implementation of **alternative** stack splitting strategies:
  - Horizontal splitting.
  - Diagonal splitting.
- Combining all models for supporting **clusters of multicores**:
  - Different **teams** should be assigned to different **cluster nodes** and share work performing **stack splitting**.
  - A team of workers should run on shared memory and **workers inside a team** can distribute work using **environment copying** or **stack splitting**.