Or-Parallel Prolog Execution on Clusters of Multicores

João Santos and Ricardo Rocha

CRACS & INESC TEC
Faculty of Sciences, University of Porto

SLATE'13 – June 20, 2013

Motivation

- The inherent non-determinism in the way logic programs are structured makes Prolog very attractive for the exploitation of implicit parallelism. Prolog offers two major forms of implicit parallelism:
  - Or-Parallelism
  - And-Parallelism

Motivation

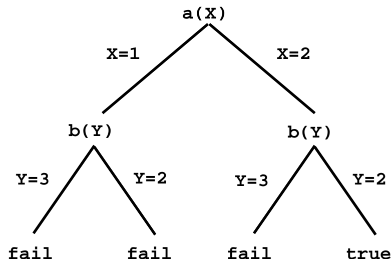
- The inherent non-determinism in the way logic programs are structured makes Prolog very attractive for the exploitation of implicit parallelism. Prolog offers two major forms of implicit parallelism:
  - Or-Parallelism
  - And-Parallelism
- Many parallel Prolog systems have been developed in the past, however none of them was specially designed to explore the combination of shared with distributed memory architectures. Arguably, the most well-know systems are based on the environment copying model:
  - Shared memory: Muse (Yap Prolog)
  - Distributed memory: PALS (Yap Prolog)

Motivation

- The inherent non-determinism in the way logic programs are structured makes Prolog very attractive for the exploitation of implicit parallelism. Prolog offers two major forms of implicit parallelism:
  - Or-Parallelism
  - And-Parallelism
- Many parallel Prolog systems have been developed in the past, however none of them was specially designed to explore the combination of shared with distributed memory architectures. Arguably, the most well-know systems are based on the environment copying model:
  - Shared memory: Muse (Yap Prolog)
  - Distributed memory: PALS (Yap Prolog)
- Our goal is to design, develop and implement a novel computational model (on top of the Yap Prolog system) to efficiently exploit implicit or-parallelism from the recent architectures based on clusters of multicores.

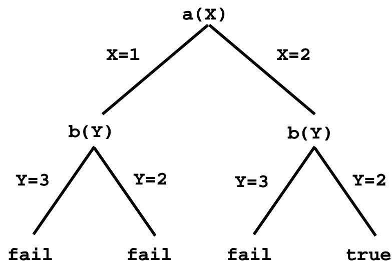
Prolog and SLD Resolution

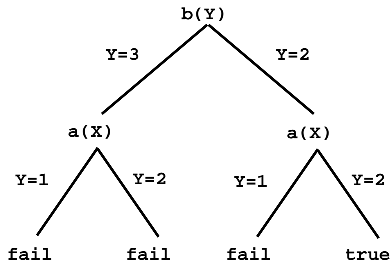
```
?- a(X), b(Y), X==Y.

a(1).            b(3).
a(2).            b(2).
```



Prolog and SLD Resolution

```
?- a(X), b(Y), X==Y.

a(1).          b(3).
a(2).          b(2).
```





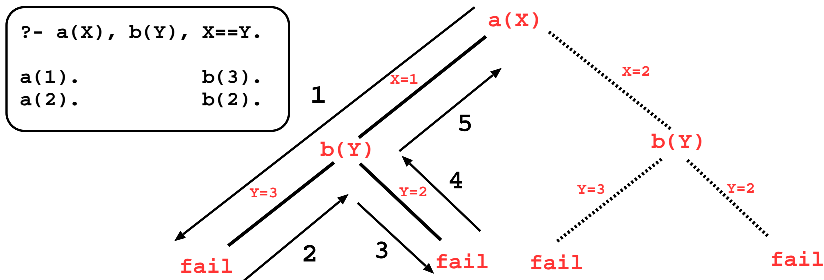
Prolog and SLD Resolution

```
sld_resolution(Query)
   select_subgoal(Query, B)
   do
      select_clause(Program, (Head :- Body))
      if (mgu = most_general_unifier(B, Head))
         Query = assign(mgu, Query - B + Body)
         if (Query)
            sld_resolution(Query)
         else
            SUCCESS
   until (no clauses left)
   FAIL
```

Prolog and SLD Resolution

In Prolog, SLD resolution is applied in following way:

- select_subgoal(): from left to right
- select_clause(): top-down



Implicit And-Parallelism

```
sld_resolution(Query)
   select_literal(Query, B) // and-parallelism
   do
      select_clause(Program, (Head :- Body))
      if (mgu = most_general_unifier(B, Head))
         Query = assign(mgu, Query - B + Body)
         if (Query)
            sld_resolution(Query)
         else
            SUCCESS
   until (no clauses left)
   FAIL
```

Implicit Or-Parallelism

```
sld_resolution(Query)
   select_literal(Query, B)
   do
      select_clause(Program, (Head :- Body)) //or-parallelism
      if (mgu = most_general_unifier(B, Head))
         Query = assign(mgu, Query - B + Body)
         if (Query)
            sld_resolution(Query)
         else
            SUCCESS
   until (no clauses left)
   FAIL
```

Multiple Bindings Problem

When implementing or-parallelism, a main difficulty is how to efficiently represent the multiple bindings for the same variable produced by the parallel execution of alternative matching clauses.

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

Multiple Bindings Problem

When implementing or-parallelism, a main difficulty is how to efficiently represent the multiple bindings for the same variable produced by the parallel execution of alternative matching clauses.

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

creation of var X

Multiple Bindings Problem

When implementing or-parallelism, a main difficulty is how to efficiently represent the multiple bindings for the same variable produced by the parallel execution of alternative matching clauses.

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

creation of var X

?- a(X)

..., X=3, ...

worker 0

..., X=5, ...

worker 1

Multiple Bindings Problem: Binding Arrays

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

```
?- a(X)
```

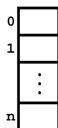
Multiple Bindings Problem: Binding Arrays

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

counter=0

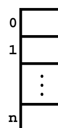
?- a(X)

Binding Array



worker 0

Binding Array

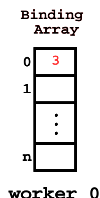


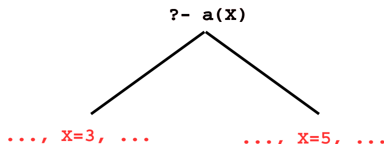
worker 1

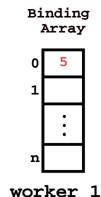
Multiple Bindings Problem: Binding Arrays

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

counter=1







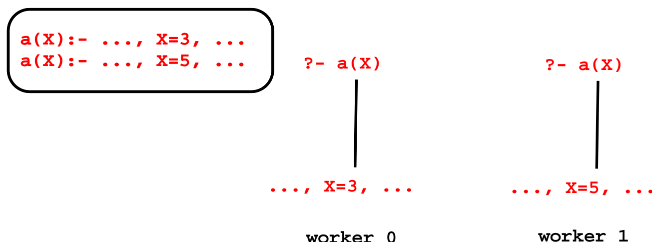
Multiple Bindings Problem: Environment Copy

With environment copying model, each worker keeps a separate copy of its own environment, thus the bindings to shared variables are done as usual and without conflicts.

```
a(X):- ..., X=3, ...
a(X):- ..., X=5, ...
```

Multiple Bindings Problem: Environment Copy

With environment copying model, each worker keeps a separate copy of its own environment, thus the bindings to shared variables are done as usual and without conflicts.



Advantages:

- each worker acts like an independent Prolog machine
- requires minimal synchronization

Scheduling and Environment Copying

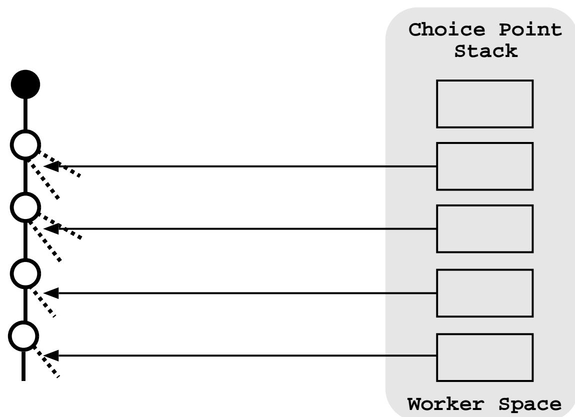
Two of the most successful systems using environment copying are:

- Muse (shared memory architectures)
- PALS (distributed memory architectures)

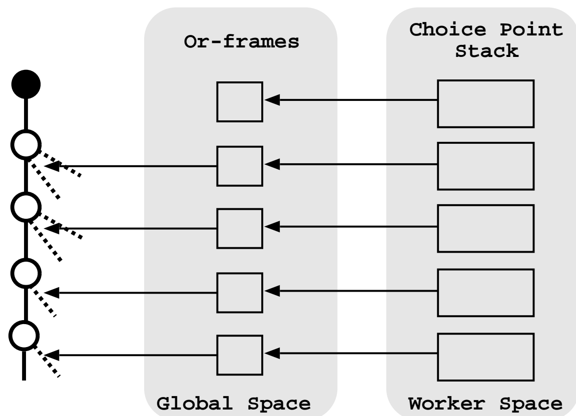
The main difference between both systems is on how scheduling is done:

- Muse uses dynamic distribution of work (or-frames)
- PALS uses static distribution of work (stack splitting)

Dynamic Distribution of Work: Or-Frames

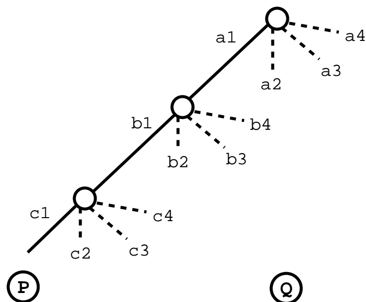


Dynamic Distribution of Work: Or-Frames



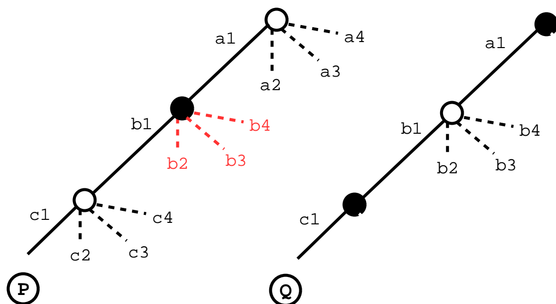
Static Distribution of Work: Stack Splitting

Before sharing



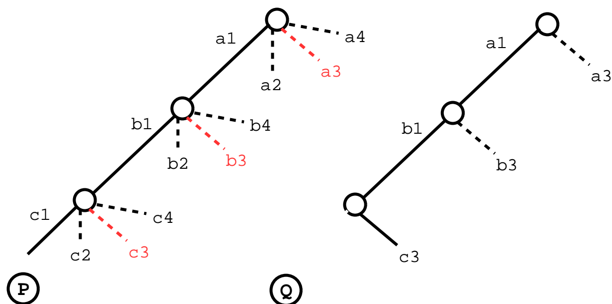
Static Distribution of Work: Stack Splitting

After sharing (vertical splitting)



Static Distribution of Work: Stack Splitting

After sharing (horizontal splitting)



Our Proposal

The goal behind our proposal is to implement the concept of teams and the ability to exploit different scheduling strategies for distributing work among teams and among the workers inside a team (two-level scheduler).

Our Proposal

The goal behind our proposal is to implement the concept of teams and the ability to exploit different scheduling strategies for distributing work among teams and among the workers inside a team (two-level scheduler).

We define a team as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem.

Our Proposal

The goal behind our proposal is to implement the concept of teams and the ability to exploit different scheduling strategies for distributing work among teams and among the workers inside a team (two-level scheduler).

We define a team as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem.

Since all workers inside a team share the same address space this implies that all workers should be in the same computer node. On the other hand, we also want to be possible to have several teams in a computer node or distributed by other nodes.

Our Proposal

For (shared memory) multicores, we can thus have any combination of strategies, and distribute work using both dynamic or static scheduling of work.

Our Proposal

For (shared memory) multicores, we can thus have any combination of strategies, and distribute work using both dynamic or static scheduling of work.

For (distributed memory) clusters of multicores, we can only have static scheduling of work for distributing work among teams, but we can still have dynamic or static scheduling of work for distributing work among the workers inside a team.

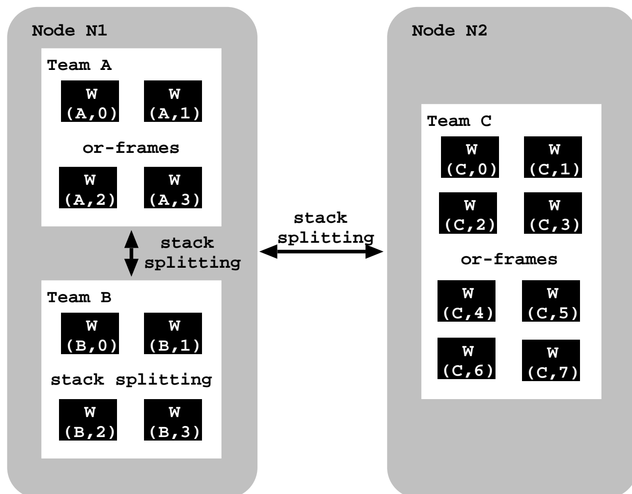
Our Proposal

For (shared memory) multicores, we can thus have any combination of strategies, and distribute work using both dynamic or static scheduling of work.

For (distributed memory) clusters of multicores, we can only have static scheduling of work for distributing work among teams, but we can still have dynamic or static scheduling of work for distributing work among the workers inside a team.

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.

Our Proposal



Work Sharing Process

To distribute work inside a team, we can use, with minor adaptations, any of Yap's current dynamic or static schedulers for shared memory.

Work Sharing Process

To distribute work inside a team, we can use, with minor adaptations, any of Yap's current dynamic or static schedulers for shared memory.

For work sharing among teams, our approach is to implement a layered approach, and for that a second-level scheduler will be used.

- We will only ask for work to other teams when no more work exists in a team.
- However, the sharing process between teams will still be done between two workers.
- The selected worker of the idle team is then the responsible for sharing the new work with its teammates.

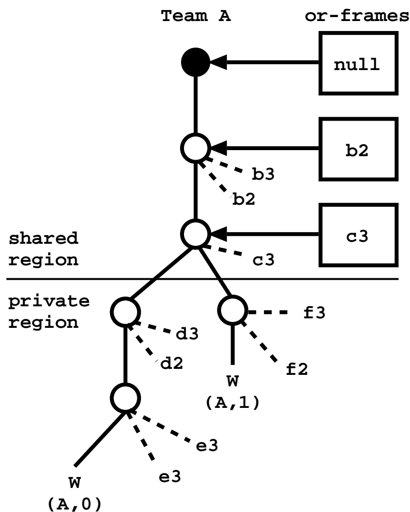
Work Sharing Process

```
WorkerGetWork(W,T)
   while (TeamNotFinished(T))
      while (TeamWithWork(T))
         B = SelectBusyWorker(T)
         if (SendShareRequest(W,B) = ACCEPTED)
            ShareWork(W,B)
            return TRUE
      if (W = SelectMasterWorker(T))
         if (TeamGetWork(W,T))
            return TRUE
         else
            SetTeamAsFinished(T)
   return FALSE
```

Work Sharing Process

```
TeamGetWork(W,T)
   while (not AllTeamsWithoutWork())
      U = SelectBusyTeam()
      if (SendShareRequest(T,U) = ACCEPTED)
         S = GetSharingWorker(U)
         ShareWork(W,S)
         return TRUE
   return FALSE
```

Second-Level Scheduler



Second-Level Scheduler

Team A

or-frames

null

null

c3

null

b2

null

d2

b3

b2

shared region

c3

private region

d3

d2

f3

f2

W (A,1)

e3

e3

W (A,0)

Second-Level Scheduler

Team A

or-frames

null

null

c3

b3

b2

shared region

c3

private region

d3

d2

f3

f2

W (A,1)

e3

e3

W (A,0)

null

b2

null

d2

Team C

b3

b2

d3

W (C,0)

Current State and Future Work

Current state:

- We have a new memory layout to deal with teams
- Most code is already adapted to deal with teams
- Currently, it is possible to have two teams with one worker each sharing work

Current State and Future Work

Current state:

- We have a new memory layout to deal with teams
- Most code is already adapted to deal with teams
- Currently, it is possible to have two teams with one worker each sharing work

Future Work:

- Finish the first running version of our proposal
- Study alternative protocols to implement the sharing work process between teams
- Explore different load balancing strategies in order to achieve optimal resource utilization when distributing work across teams
- Avoid speculative work