YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters

Ricardo Rocha Fernando Silva Rolando Martins
{*ricroc,fds,rolando*}*@ncc.up.pt*

DCC-FC & LIACC
University of Porto, Portugal

Why Parallelism?

Why Parallelism?

➤ Performance

- The ability to speedup Prolog execution is fundamental for real world applications.
- Better performance can be achieved by:
  - Improving the efficiency of sequential implementations.
  - Developing efficient parallel execution models.

Why Parallelism?

Performance

- The ability to speedup Prolog execution is fundamental for real world applications.
- Better performance can be achieved by:
  - Improving the efficiency of sequential implementations.
  - Developing efficient parallel execution models.

Implicit Parallelism

- Prolog's execution model allows parallelism to be exploited implicitly, without extra input from the programmer to express or manage parallelism.
- This makes parallel logic programming as easy as logic programming.

Main Forms of Implicit Parallelism

And-Parallelism

- It appears when more than one subgoal is present in the query or in the body of a clause. It corresponds to the parallel execution of such subgoals.

```
a(X,Y) :- b(X,Z), c(Z,Y).
```

Main Forms of Implicit Parallelism

And-Parallelism

- It appears when more than one subgoal is present in the query or in the body of a clause. It corresponds to the parallel execution of such subgoals.

```
a(X,Y) :- b(X,Z), c(Z,Y).
```

Or-Parallelism

- It appears when a subgoal call unifies with more than one of the clauses defining the subgoal predicate. It corresponds to the parallel execution of the bodies of alternative matching clauses.

```
a(X,Y) :- b(X), c(Y).
a(X,Y) :- d(X,Y), e(Y).
a(X,Y) :- f(X,Z), g(Z,Y).
```

Main Forms of Implicit Parallelism

And-Parallelism

- It appears when more than one subgoal is present in the query or in the body of a clause. It corresponds to the parallel execution of such subgoals.

```
a(X,Y) :- b(X,Z), c(Z,Y).
```

Or-Parallelism

- It appears when a subgoal call unifies with more than one of the clauses defining the subgoal predicate. It corresponds to the parallel execution of the bodies of alternative matching clauses.

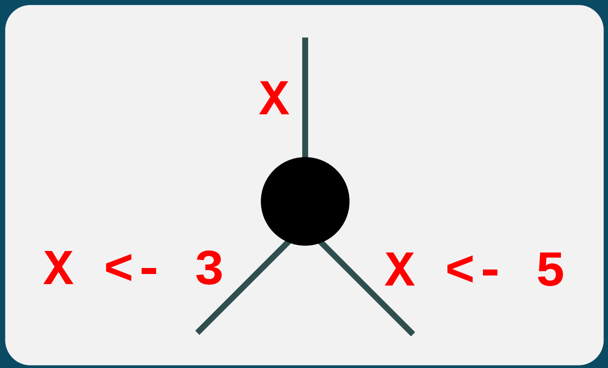
```
a(X,Y) :- b(X), c(Y).
a(X,Y) :- d(X,Y), e(Y).
a(X,Y) :- f(X,Z), g(Z,Y).
```

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

Or-Parallelism: Main Problems

➤ Multiple Bindings

- Alternative branches have to be organized in such a way that conflicting bindings for shared variables can be easily discernible.

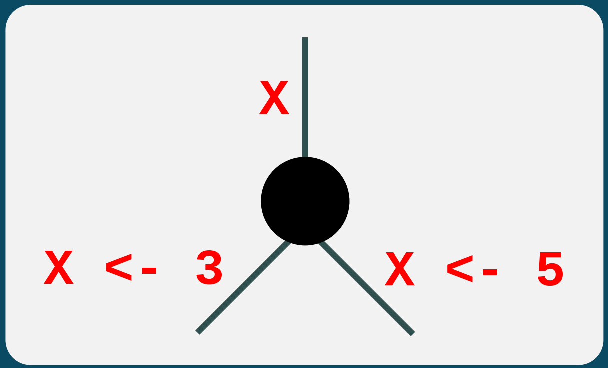


- Private areas to store the bindings for each branch are required.

Or-Parallelism: Main Problems

➤ Multiple Bindings

- Alternative branches have to be organized in such a way that conflicting bindings for shared variables can be easily discernible.



- Private areas to store the bindings for each branch are required.

➤ Scheduling

- Work scheduling is a complex problem because of the dynamic nature of work in or-parallel systems, as in fact, unexploited branches arise irregularly.
- Careful scheduling strategies are required.

The YapDss System

- **Our Goal:** design and implement an Or-Parallel Prolog system for a new type of distributed memory platforms, the **Beowulf PC clusters**.
  - Build from off-the-shelf components
  - Low-cost
  - Scalable
  - Viable alternative to traditional shared memory platforms

The YapDss System

- **Our Goal:** design and implement an Or-Parallel Prolog system for a new type of distributed memory platforms, the **Beowulf PC clusters**.
  - Build from off-the-shelf components
  - Low-cost
  - Scalable
  - Viable alternative to traditional shared memory platforms
- **Our Approach:** extend the Yap Prolog system to support stack splitting, a refined version of the environment copying model.

The YapDss System

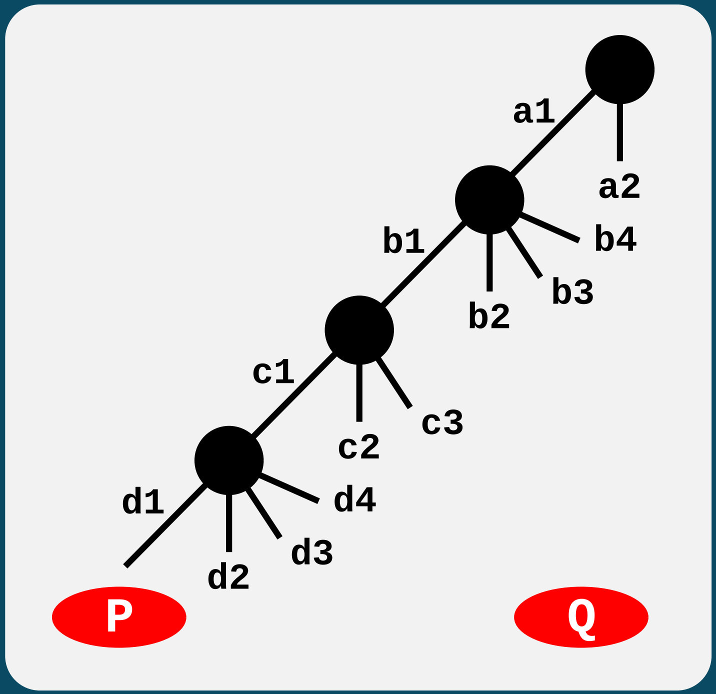
- **Our Goal:** design and implement an Or-Parallel Prolog system for a new type of distributed memory platforms, the **Beowulf PC clusters**.
  - Build from off-the-shelf components
  - Low-cost
  - Scalable
  - Viable alternative to traditional shared memory platforms
- **Our Approach:** extend the Yap Prolog system to support stack splitting, a refined version of the environment copying model.
  - In copying, sharing is done by **copying the execution stacks** between workers. To avoid redundant computations this requires further synchronization.
  - Stack splitting (PALS system) introduces a heuristic that when sharing, **work is split beforehand**, in such a way that no further synchronization is needed.

The YapDss System: Main Contributions

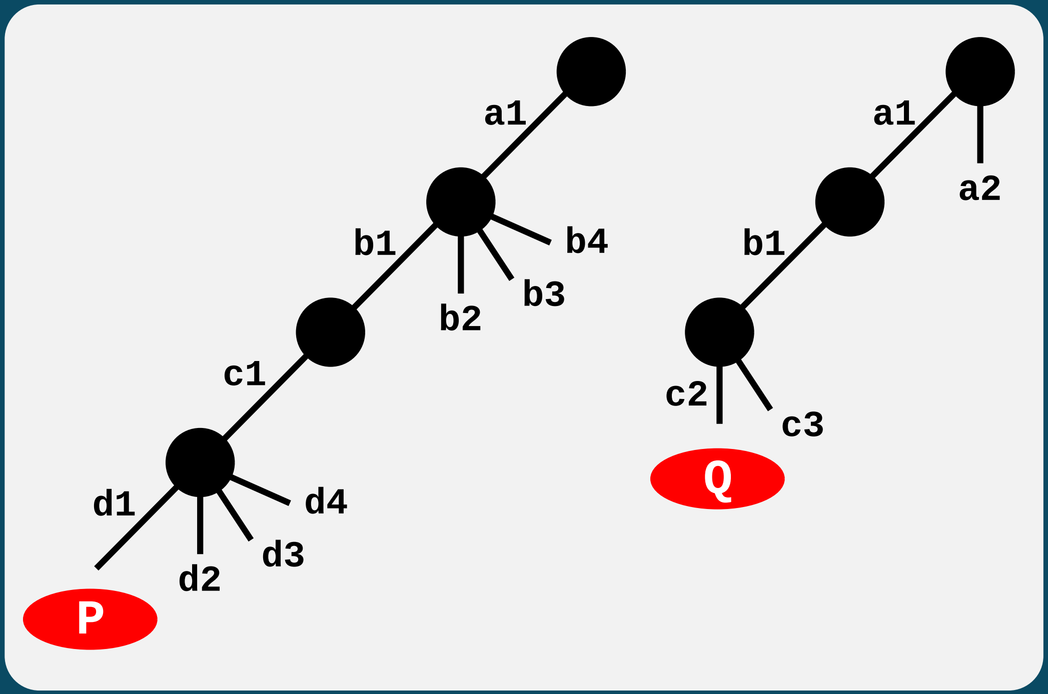
- **Diagonal Stack Splitting**
  - Better work load balance among the computing workers.
- **Branch Array**
  - Simple scheme to determine the bottommost common node between the branches of two workers.
- **Work Load**
  - The work load of a worker is calculated exactly, it is not an estimate.

Vertical Stack Splitting

➤ Each worker is given all the untried alternatives in alternate choice points, starting from worker P with its current choice point.



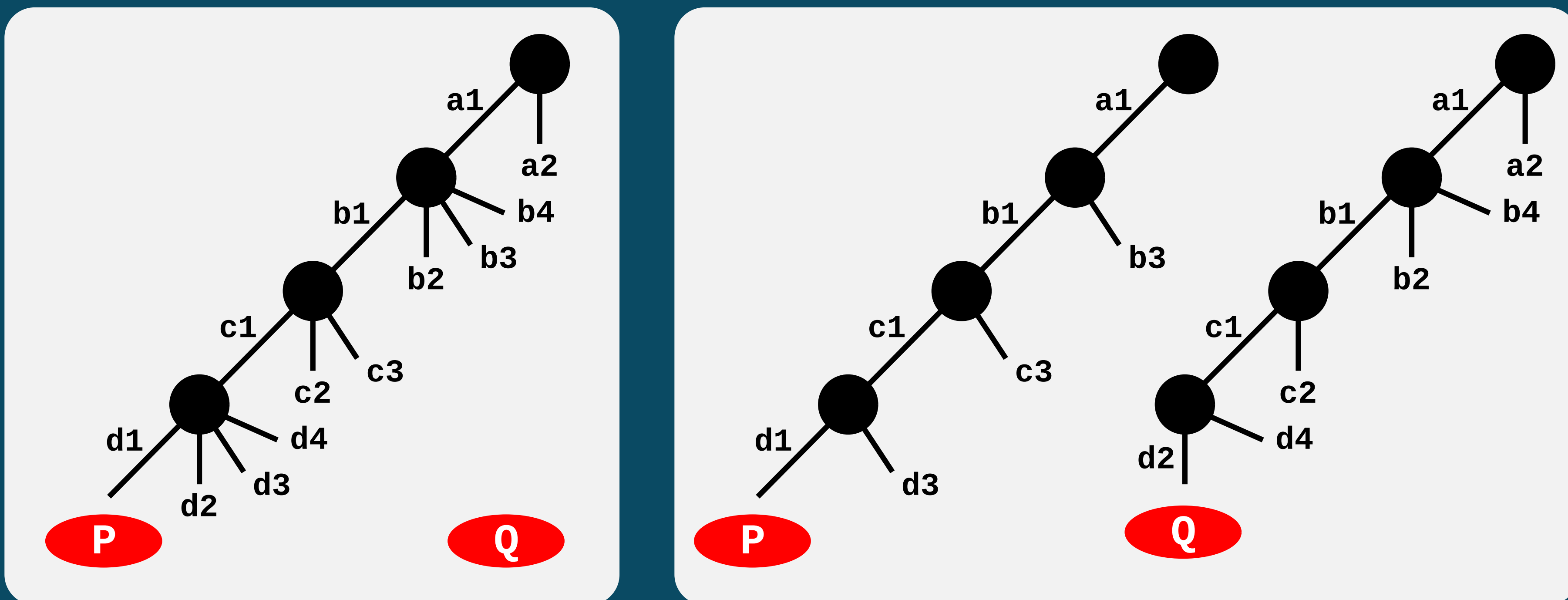
before



after vertical splitting

Horizontal Stack Splitting

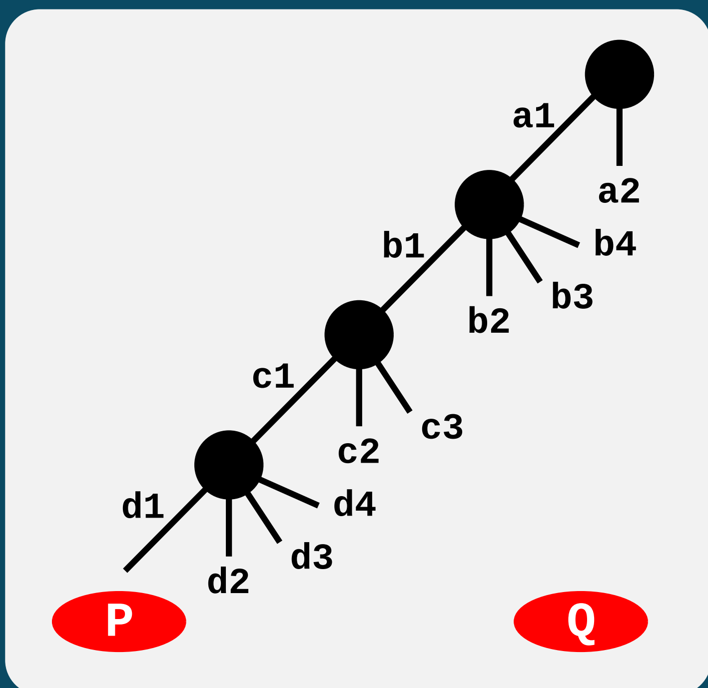
- The untried alternatives in each choice point are alternatively split between the requesting worker Q and the sharing worker P.



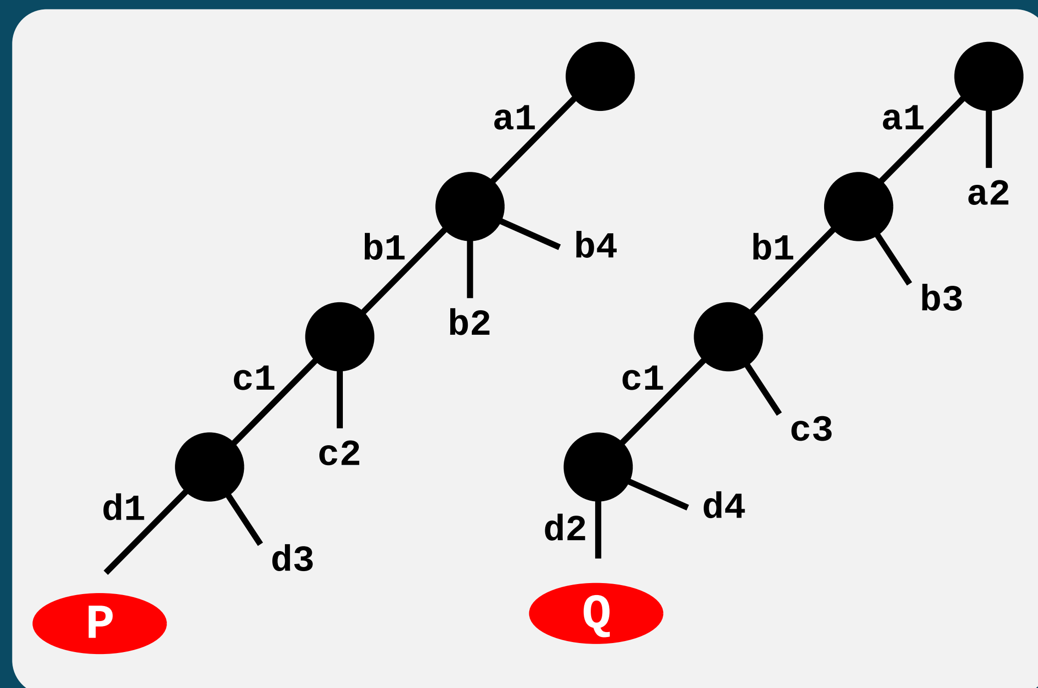
before **after horizontal splitting**

Diagonal Stack Splitting

- The set of untried alternatives in all choice points are alternatively split between both workers.



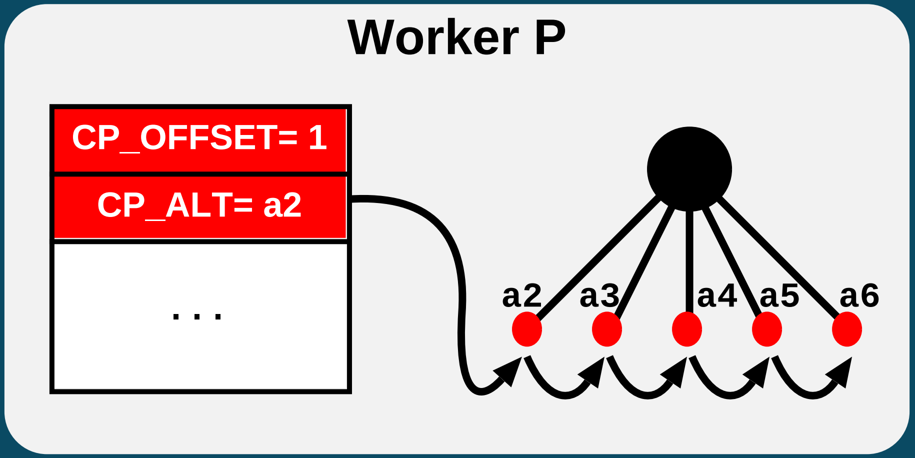
before



after diagonal splitting

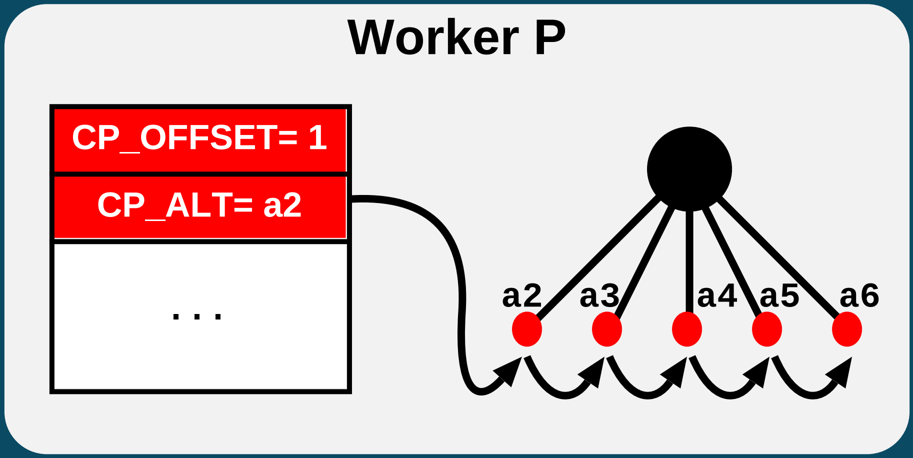
How to Split?

- Extend choice points with an extra field, CP_OFFSET, to mark the offset of the next untried alternative belonging to the choice point.
- For private choice points CP_OFFSET is always 1.



How to Split?

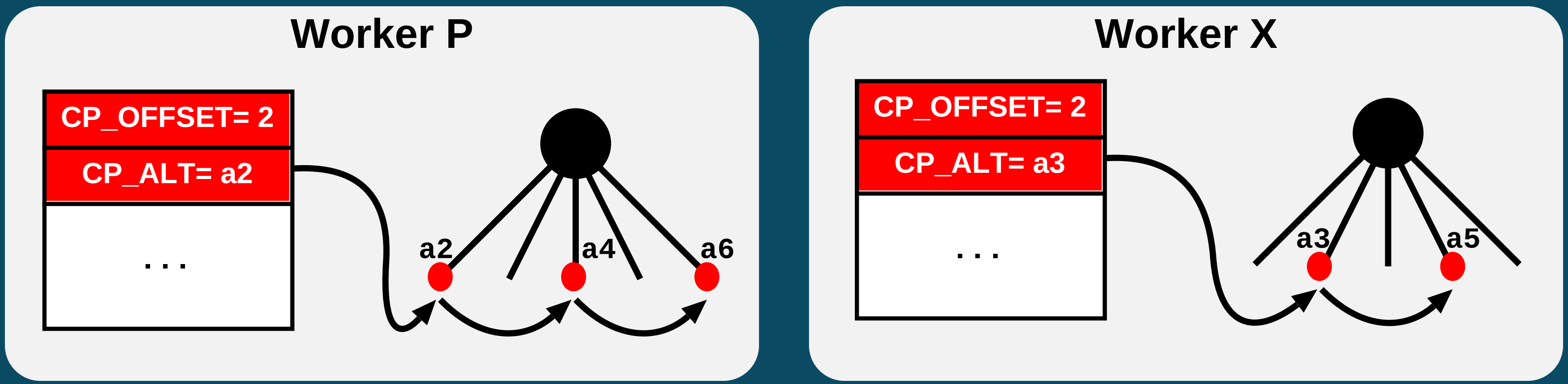
- Extend choice points with an extra field, CP_OFFSET, to mark the offset of the next untried alternative belonging to the choice point.
- For private choice points CP_OFFSET is always 1.



- When sharing a choice point we double the value in the CP_OFFSET.
- The worker that do not start the partitioning updates the CP_ALT field of its choice point to refer to the next available alternative.

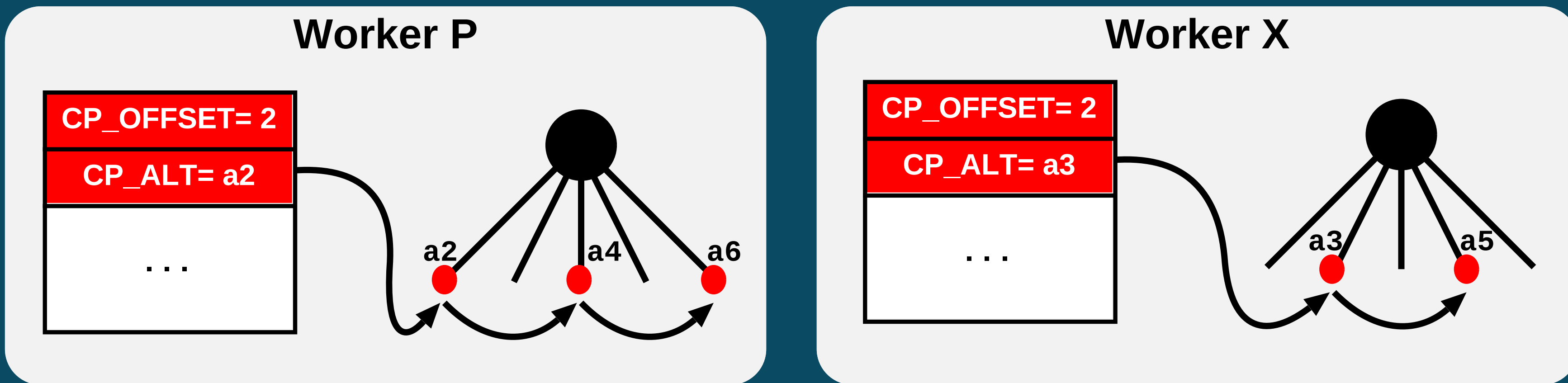
How to Split?

P sharing work with X

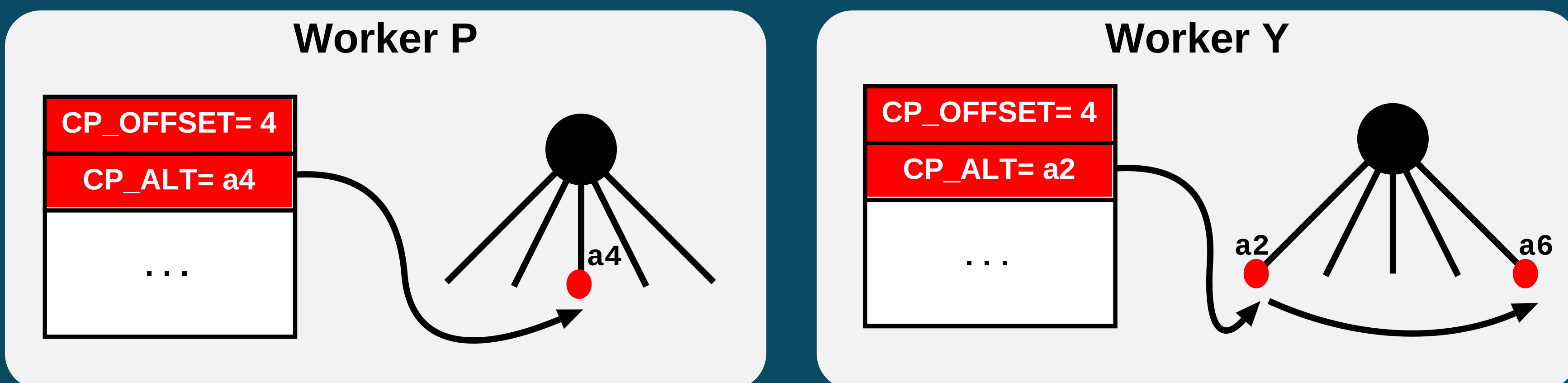


How to Split?

P sharing work with X



P sharing work with Y



How to Split?

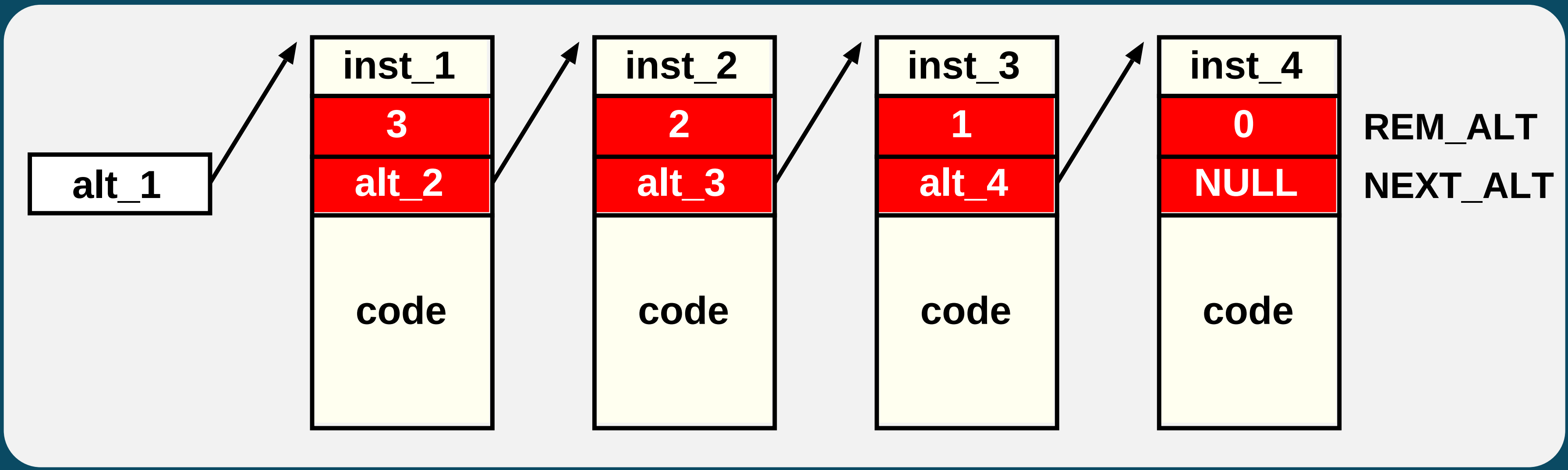
- When splitting we need to know if the number of available alternatives in a choice point is odd or even in order to decide which worker starts the partitioning in the upper choice point.

How to Split?

- When splitting we need to know if the number of available alternatives in a choice point is odd or even in order to decide which worker starts the partitioning in the upper choice point.
- A possibility is to follow the list of available alternatives and count its number.

How to Split?

- When splitting we need to know if the number of available alternatives in a choice point is odd or even in order to decide which worker starts the partitioning in the upper choice point.
- A possibility is to follow the list of available alternatives and count its number.
- YapDss takes advantage of the compiler to include information about the number of remaining alternatives starting from an alternative.



Branch Array

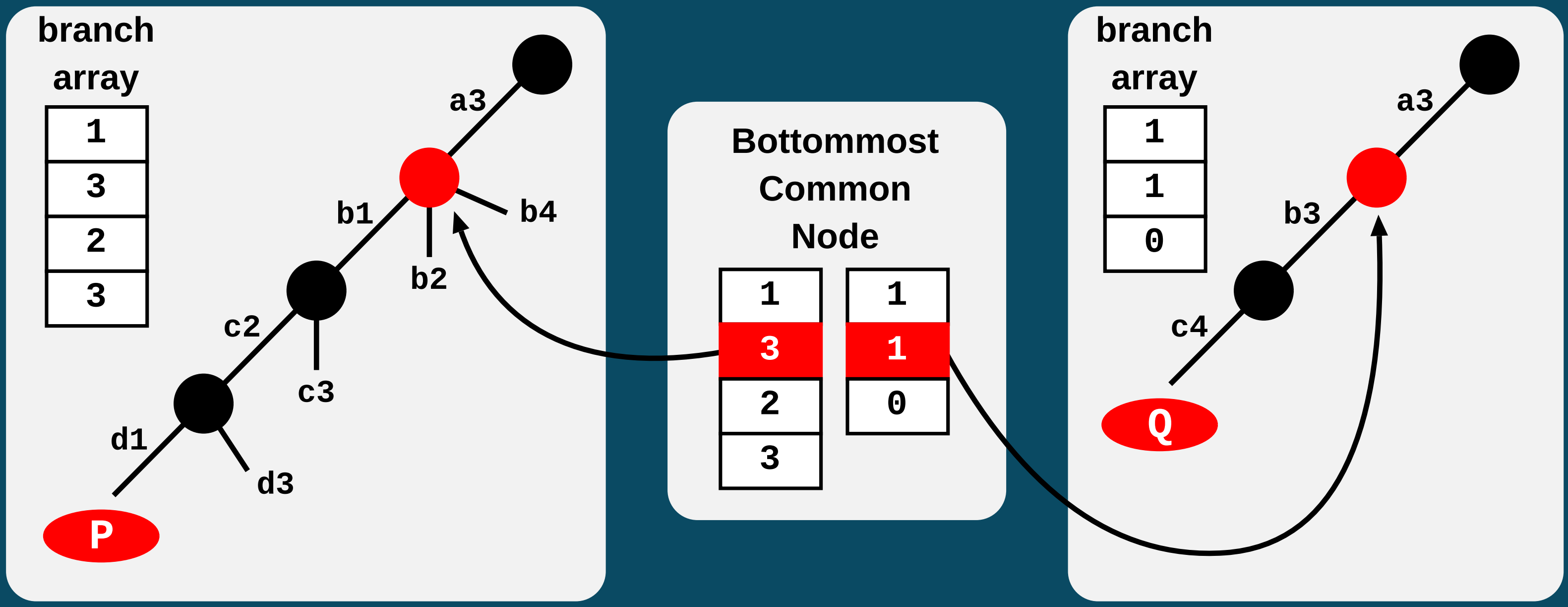
- To minimize overheads during copying we need to support incremental copying.

Branch Array

- To minimize overheads during copying we need to support incremental copying.
- To support incremental copying we need a mechanism that allows us to quickly find the bottommost common node between two workers.

Branch Array

- To minimize overheads during copying we need to support incremental copying.
- To support incremental copying we need a mechanism that allows us to quickly find the bottommost common node between two workers.
- YapDss uses a private branch array to uniquely represent the position of each worker. The depth of a choice point identifies its offset in the branch array.



Sharing Work

- Q makes a sharing request to P
  - Q sends a message to P that includes its branch array
- P decides to share work with Q
  - P calculates the bottommost common node
  - P computes the stack segments to be copied to Q
  - P packs all the information in a message and sends it back to Q
- Q receives a positive answer
  - Q copies the stack segments in the message to the proper space in its execution stacks
- P and Q apply diagonal splitting

Initial Performance Evaluation

Programs	Number of Workers			
	2	4	6	8
queens12	38.93(1.99)	19.63(3.94)	13.36(5.80)	10.12(7.66)
nsort	124.24(1.98)	63.14(3.90)	42.44(5.80)	33.06(7.45)
puzzle4x4	34.00(1.99)	17.34(3.91)	11.83(5.73)	9.41(7.20)
magic	15.50(1.99)	7.88(3.92)	5.58(5.53)	4.38(7.05)
cubes7	0.67(1.96)	0.40(3.26)	0.33(3.90)	0.23(4.80)
ham	0.17(1.75)	0.10(2.81)	0.09(3.13)	0.10(2.95)
Average	(1.94)	(3.62)	(4.98)	(6.19)

- PC cluster with 4 dual Pentium II nodes interconnected by Myrinet-SAN switches
- All benchmarks find all solutions for the problem
- YapDss is on average 16% slower than Yap

Conclusion and Further Work

- Design and implementation of YapDss
  - Diagonal stack splitting
  - Branch array
- Initial performance evaluation
  - Low overhead over sequential execution
  - Excellent speedups for applications with coarse-grained parallelism and quite good results globally
- Further work
  - More detailed system evaluation and performance tuning
  - Support speculative execution with cuts
  - Integration with the official Yap distribution