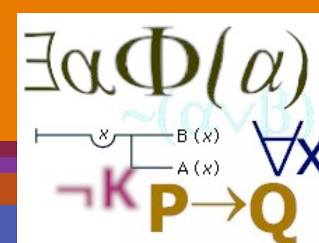
The Yesterday, Today, and Tomorrow of Parallelism in Logic Programming

Enrico Pontelli

Department of Computer Science

New Mexico State University



?-

New Mexico State University

Tutorial Roadmap

Basics

Prolog

YESTERDAY

Systems

Ciao clasp

YAProlog

ASP

TODAY

Going Large

Going Small

(EARLY) TOMORROW

Let’s get Started!

Tutorial Roadmap

Basics

Prolog

YESTERDAY

Systems

Ciao clasp

YAProlog

ASP

TODAY

Going Large

Going Small

(EARLY) TOMORROW

Prolog Programs

- Program = a bunch of axioms
- Run your program by:
  - Enter a series of facts and axioms
  - Pose a query
  - System tries to prove your query by finding a series of inference steps

- “Philosophically” declarative
- Actual implementations are deterministic

Horn Clauses (Axioms)

- Axioms in logic languages are written:

  H :- B1, B2,….,B3

Facts = clause with head and no body.

Rules = have both head and body.

Query – can be thought of as a clause with no body.

Terms

- H and B are terms.
- Terms =
  - Atoms - begin with lowercase letters: x, y, z, fred
  - Numbers: integers, reals
  - Variables - begin with captial letters: X, Y, Z, Alist
  - Structures: consist of an atom called a functor, and a list of arguments. ex. edge(a,b). line(1,2,4).

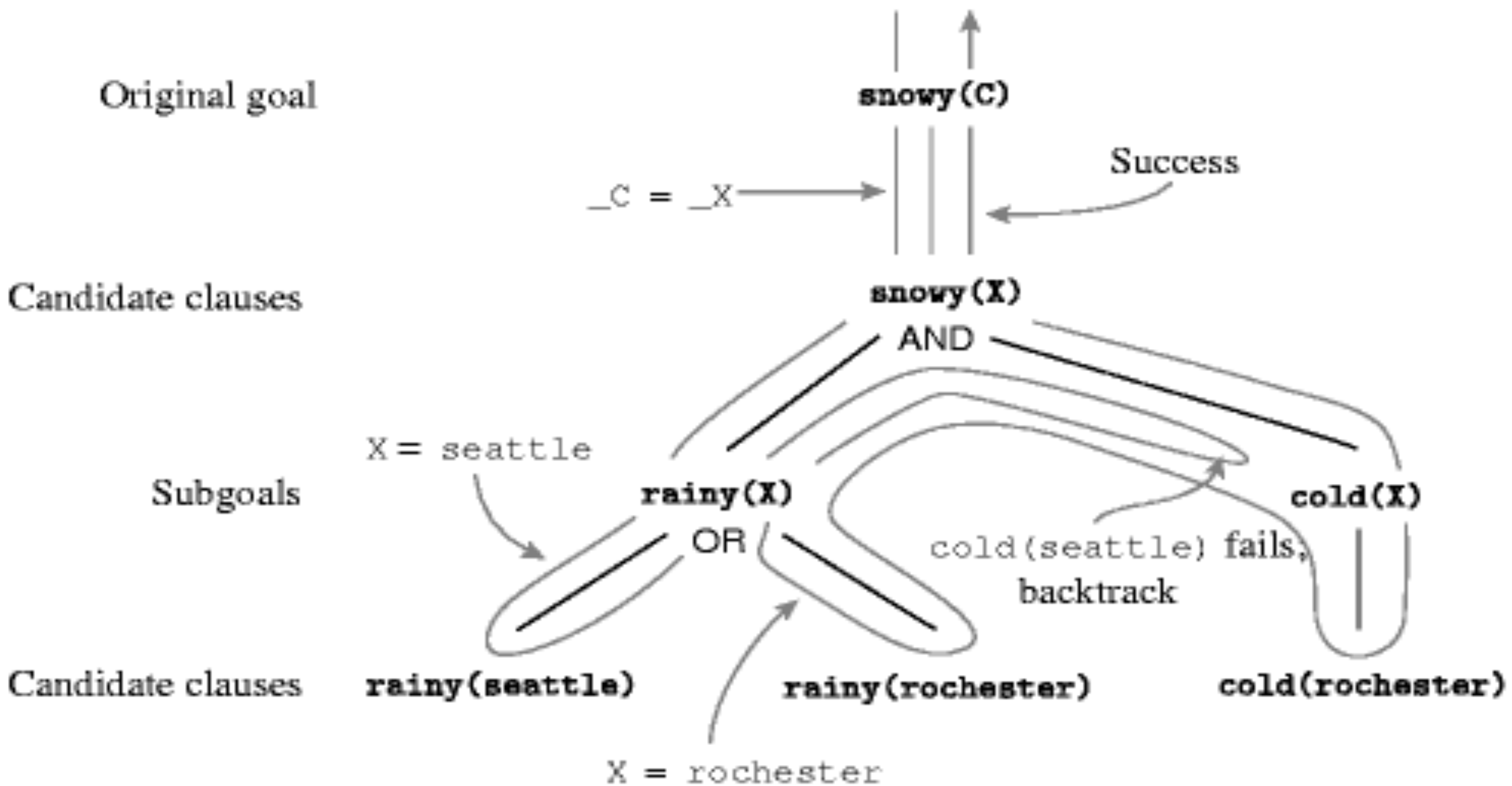
Backward Chaining

START WITH THE GOAL and work backwards, attempting to decompose it into a set of (true) clauses.

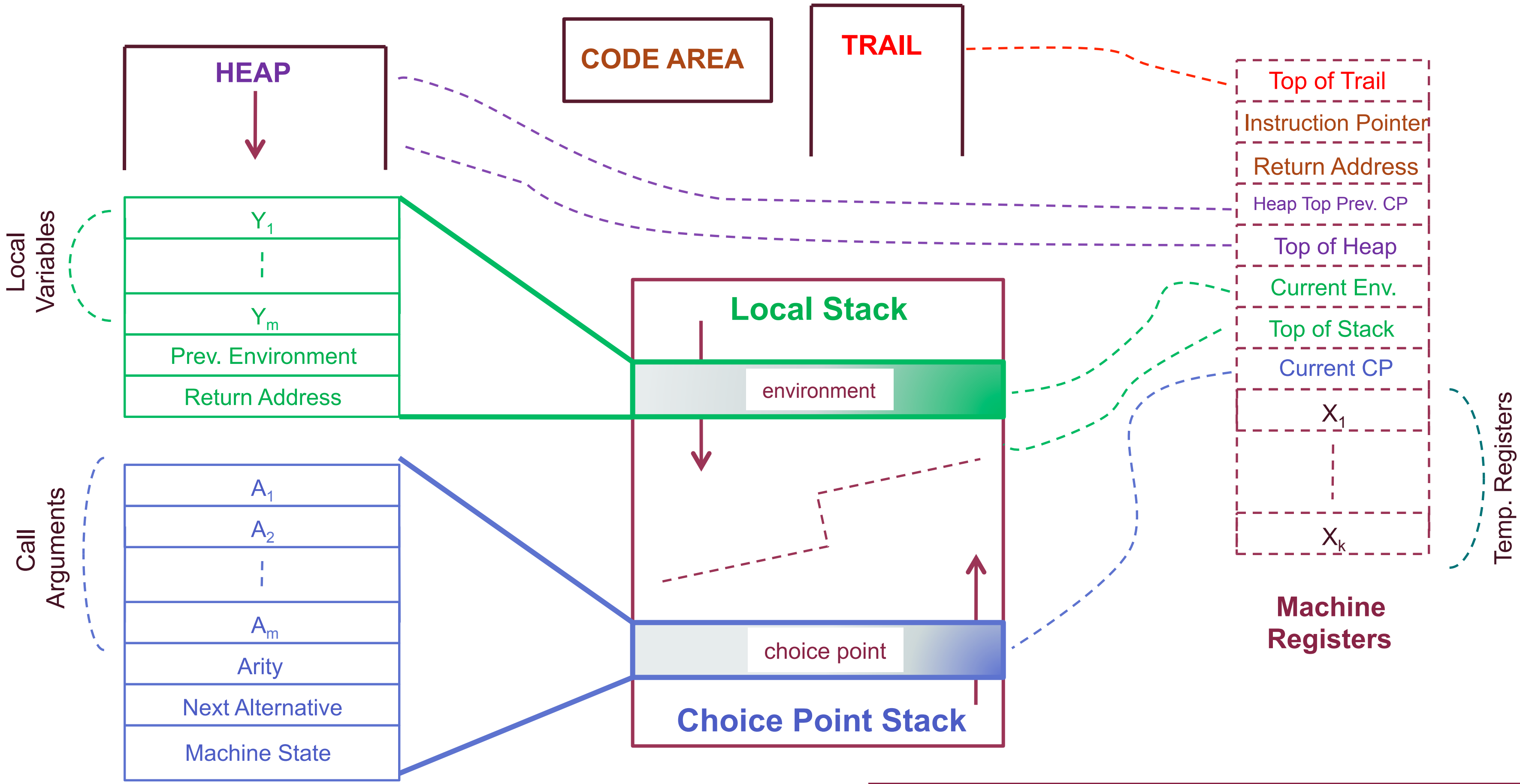
This is what the Prolog interpreter does.

Backtracking search

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```

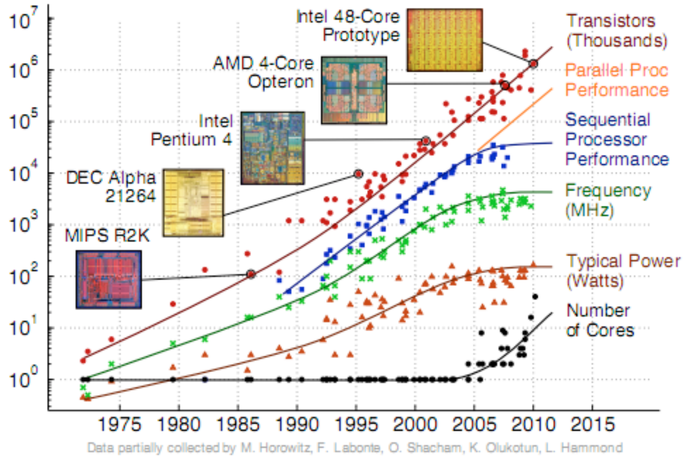


Assumption for this Tutorial

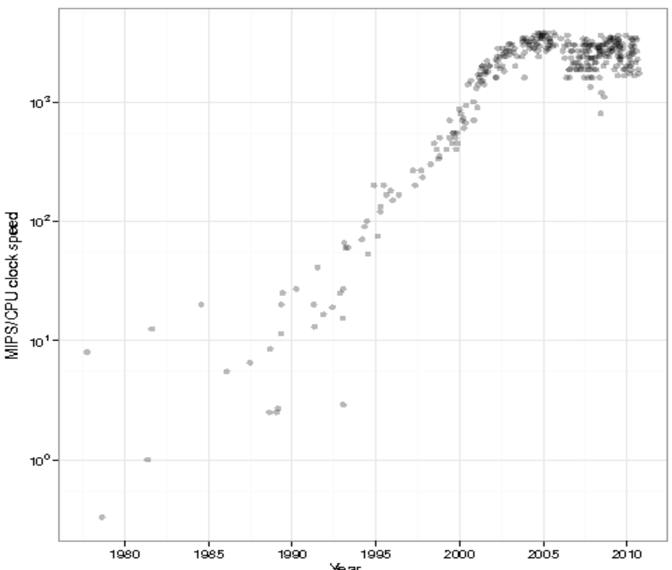


Why Parallelism?

- Thermal Wall
  - Slowing increases in clock frequency
  - Cooling
  - Power consumption



Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf



http://www.r-bloggers.com/cpu-and-gpu-trends-over-time/

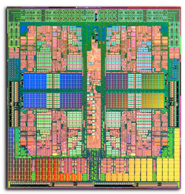
- Improve performance by placing multiple cores on the same chip

Why Parallelism?

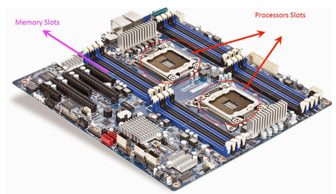
- Parallelism
  - Use multiple computational units to
    - Speed up problem resolution
    - Scale up problem resolution
- Parallel Programming is HARD!
  - High level and low level issues
    - How to partition problem, control access to resources, communication, etc.
    - How to avoid race conditions, non-determinism, latencies, Amhdahl’s law, optimize communication, etc.

Why Parallelism?

- Broad range of architectures



Copyright Advanced Micro Devices, Inc.



Source: anandtech.com

Source: NVIDIA, Inc.

Some Basic Terms from Parallel Programming

- Task:
  - Discrete section of computation (typically assigned to a processing unit)
- Scalability:
  - Capability to improve performance as more resources are made available
- Performance Measurements
  - Time
    - Sequential: T_{seq}
    - Parallel: T_n
    - Overhead: T_1/T_{seq}
    - Speedup: T_{seq} / T_n
- Granularity
  - "Size" of the tasks performed in parallel
    - Coarse: large amounts of computation between communication steps
    - Fine: small amounts of computation between communication steps

MOTIVATIONS

Motivations

- Sequential systems are highly optimized
  - E.g., highly competitive ASP systems
- Desire to apply to even more complex and challenging real-world problems
  - Ontologies and knowledge representation
  - Planning
  - Bioinformatics
  - NLP
- Pushing the limit of sequential systems

Logic Programming and Parallelism

- Interest spawned by
  - LP ⇒ Declarative Language ⇒ Limited or No Control ⇒ Limited Dependences ⇒ Easy Parallelism
  - Everlasting myth of *"LP = slow execution"*
- LP considered suited for parallel execution since its inception
  - Kowalski "Logic for Problem Solving" (1979)
  - Pollard's Ph.D. Thesis (1981)

G. Pollard. Parallel Execution of Horn Clause Programs. Ph.D. Dissertation, Imperial College, 1981.

Tutorial Roadmap

Basics

Prolog

YESTERDAY

Systems

Ciao clasp

YAProlog

ASP

TODAY

Going Large

Going Small

(EARLY) TOMORROW

Let’s talk Parallel Prolog: The Past

- Approaches
  - Explicit Schemes
    - *Message passing primitives* (e.g., Delta-Prolog)
    - *Blackboard primitives* (e.g., Jinni, CIAO Prolog)
    - *Dataflow/guarded languages* (e.g., KLIC)
    - Multi-threading (e.g., any modern Prolog system)
  - **Implicit (or mostly implicit) Schemes**

Models of Parallelism

```
while (Query not empty) do
    select_literal B from Query                          And-Parallelism
    repeat
        select_clause (H:-Body) from Program             Or-Parallelism

    until ( unify(H,B)  or no clauses left)              Unification Parallelism
    if (no clauses left) then FAIL
    else
        σ = MostGeneralUnifier(H,B)
        Query = ((Query \ {B}) ∪ Body)σ
    endif
endwhile
```

Unification Parallelism

- Parallelize term-reduction stage of unification

$$f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n) \mapsto \begin{pmatrix} t_1 = s_1 \\ \vdots \\ t_n = s_n \end{pmatrix}$$

- Not a major focus
  - fine grained
  - dependences – common variables
  - SIMD algorithms (e.g., Barklund)

Barklund, J., Parallel Unification . Ph.D. thesis. Uppsala Theses in Computing Science No. 9/90.
Vitter, S. and Simons, R. Parallel Algorithms for Unification and Other Complete Problems in P. ACM Annual Conference, 1984.

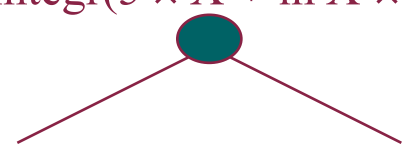
OR-PARALLELISM

Or-parallelism: Principles

- concurrent execution of different clauses unifying with a given subgoal
  - or-tree
  - or-agents

integr(X + Y, A + B) :- integr(X,A), integr(Y,B).
integr(X + Y, A $\times$ B) :- X = A1 $\times$ B, Y = A $\times$ B1, integr(A,A1), integr(B,B1).

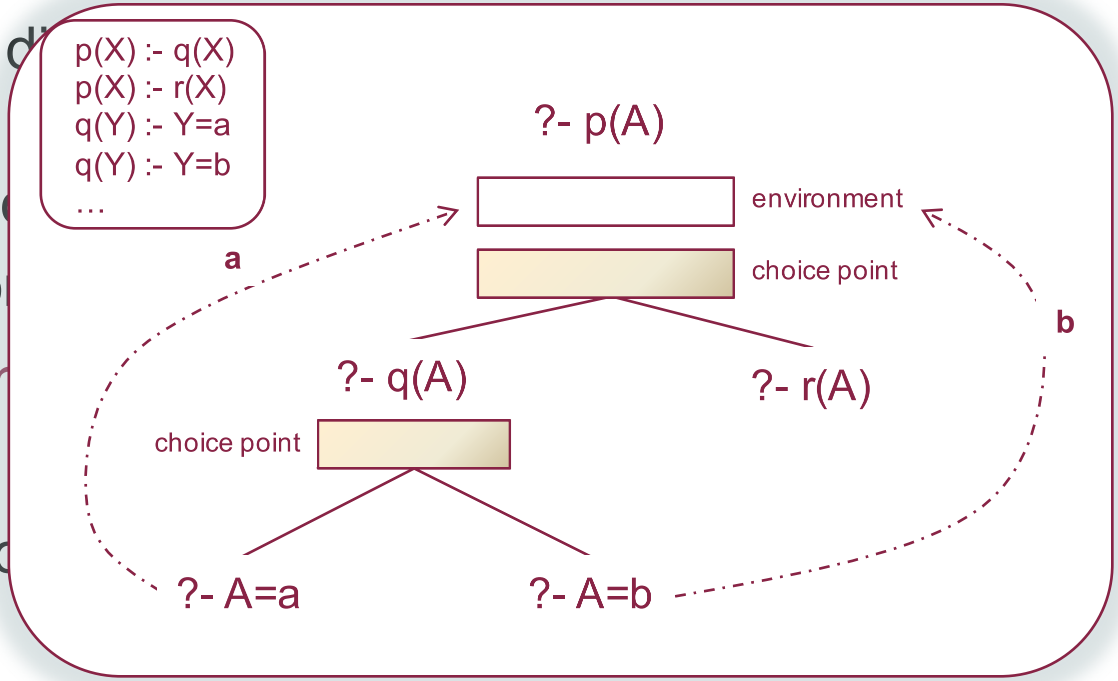
?- integr(5 $\times$ X + ln X $\times$ X, Z).



- different threads compute *different* solutions: need to be kept independent

Or-Parallelism

- Parallelize *"don't known"* non-determinism in selecting matching clauses
  - Tasks correspond to di
    Tree
  - Processes exploring
  - Computations are (fo
- Environment represen
  - Conditional variables
  - At minimum: each pr
    conditional variables



Complexity

- Abstraction of the Or-parallel Execution as operation on abstract data structures
- Program execution: construction of a (binary) labeled tree
  - `create_tree`(γ)
  - `expand`(u,γ_1,γ_2)
  - `remove(u)`
- $u \leq v$ iff u is an ancestor of v
- variables: attributes of tree nodes
- Additional operations:
  - `assign(X,u)`
  - `dereference(X,u)`
  - `alias(X1,X2,u)`

Complexity

- Restriction on assignments: for any two distinct nodes u,v such that u ≤ v there are no `assign(X,u)` and `assign(X,v)`
- Problem: $\mathcal{OP}$ problem maintaining efficiently all these operations (on-line)
- Complexity of the $\mathcal{OP}$ problem studied on pointer machines

Complexity Results

- ***Lower Bound:***

Th.: The worst-case time complexity for the $\mathcal{OP}$ problem on pointer machines is

$$\Omega(\lg N)$$

per operation.

General Idea: a pointer machine allows to access only a "small" number of records in a constant number of steps.

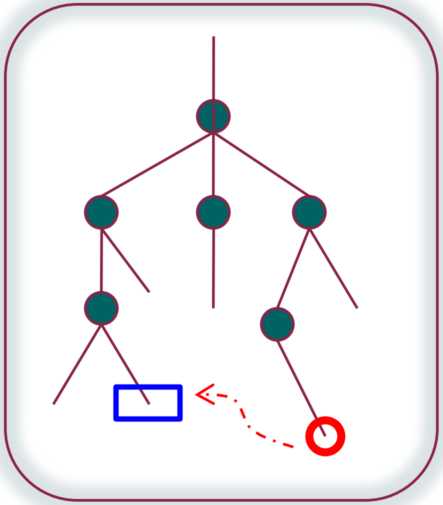
Complexity

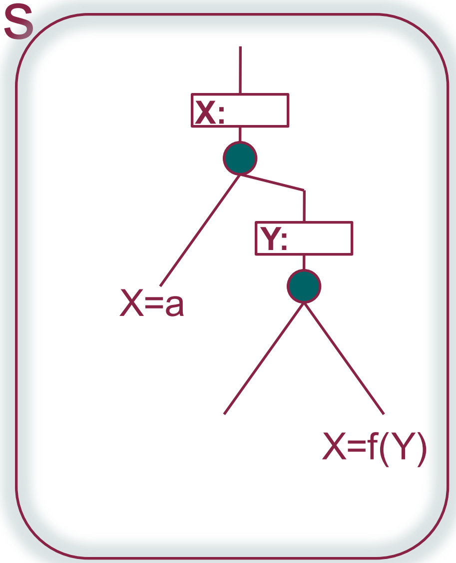
- Notes:
  - complexity results independent from the presence of aliasing (aliasing can be shown to be equivalent to union-find)
  - complexity results independent from `remove` operation (can be handled in constant-time)
- Rather large distance in complexity.
- Comparison:

Method	*Complexity*
Best Known	$O(M\ N^{1/3})$
Stack Copying	$O(M\ N)$
Binding Arrays	$O(M\ N)$
Directory Tree	$O(M\ N \lg N)$

Or-Parallelism: Classification Schemes

- Formalized in terms of three basic operations
  - binding management scheme
  - task switching scheme
  - task creation scheme
- Binding Scheme
  - Shared-tree methods
  - Non shared-tree methods
- Task Switching Scheme
  - copying schemes
  - recomputation schemes



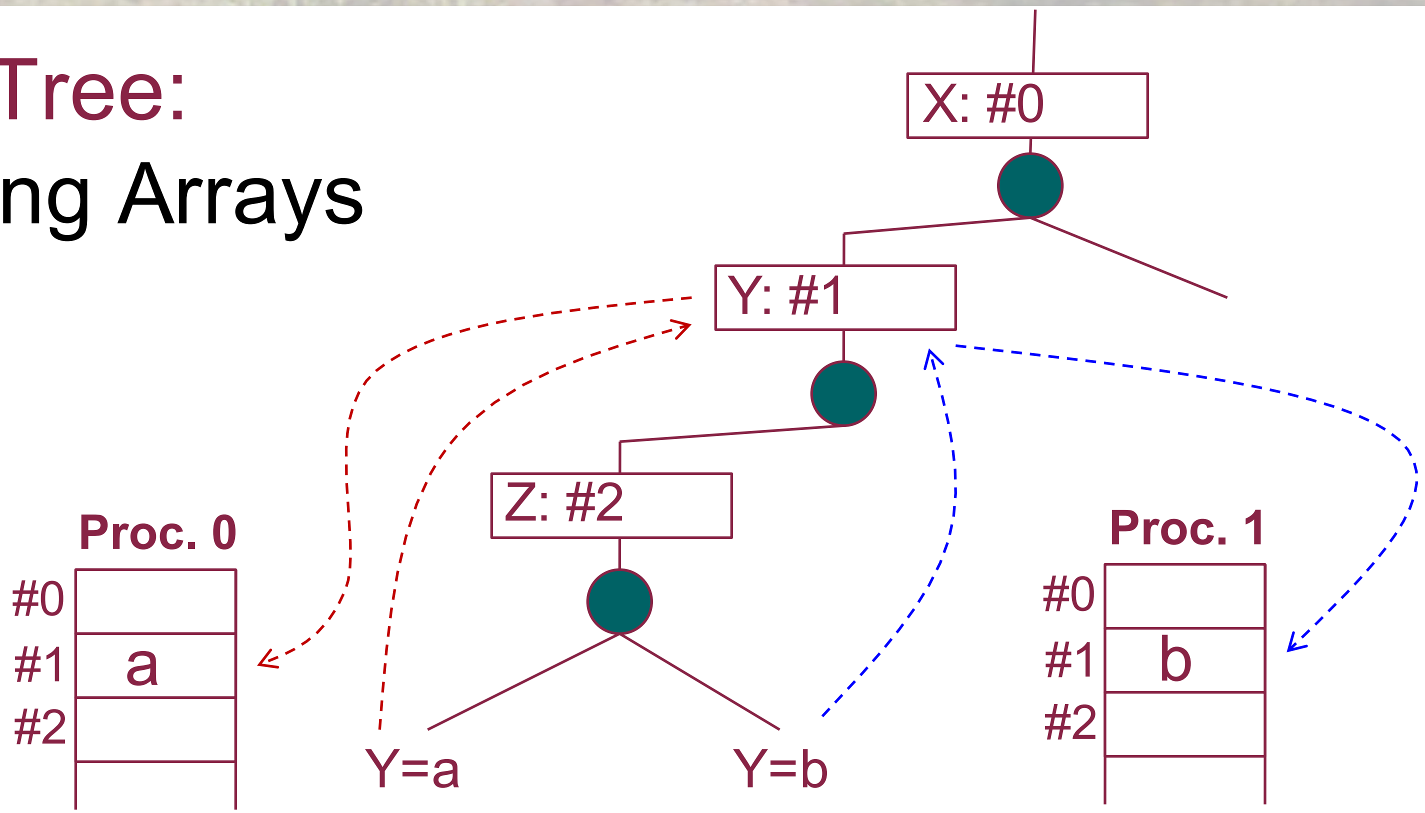


Gupta, G. and Jayaraman, B. 1993a. Analysis of Or-parallel Execution Models. ACM Transactions on Programming Languages and Systems 15, 4, 659–680.
Ranjan, D., Pontelli, E., and Gupta, G. 1999. On the Complexity of Or-Parallelism. New Generation Computing 17, 3, 285–308.

Or-Parallelism: Binding Schemes

- Shared Tree:

  Binding Arrays



Warren, D. H. D. 1987b. OR-Parallel Execution Models of Prolog. In Proceedings of TAPSOFT, Lecture Notes in Computer Science. Springer-Verlag, 243–259.

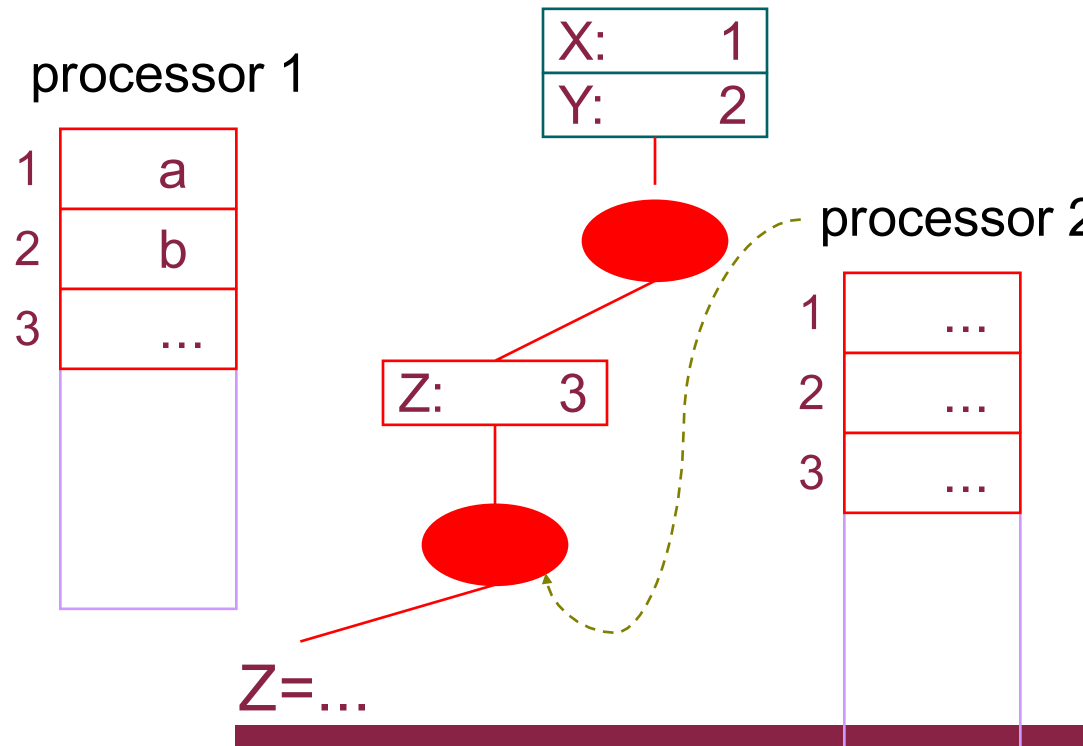
Binding Array

- constant-time variable access (one level of indirection)
- non constant-time task switching:

p(X,Y) :- … X=a,.. q(Y,Z) …
p(X,Y) :- …

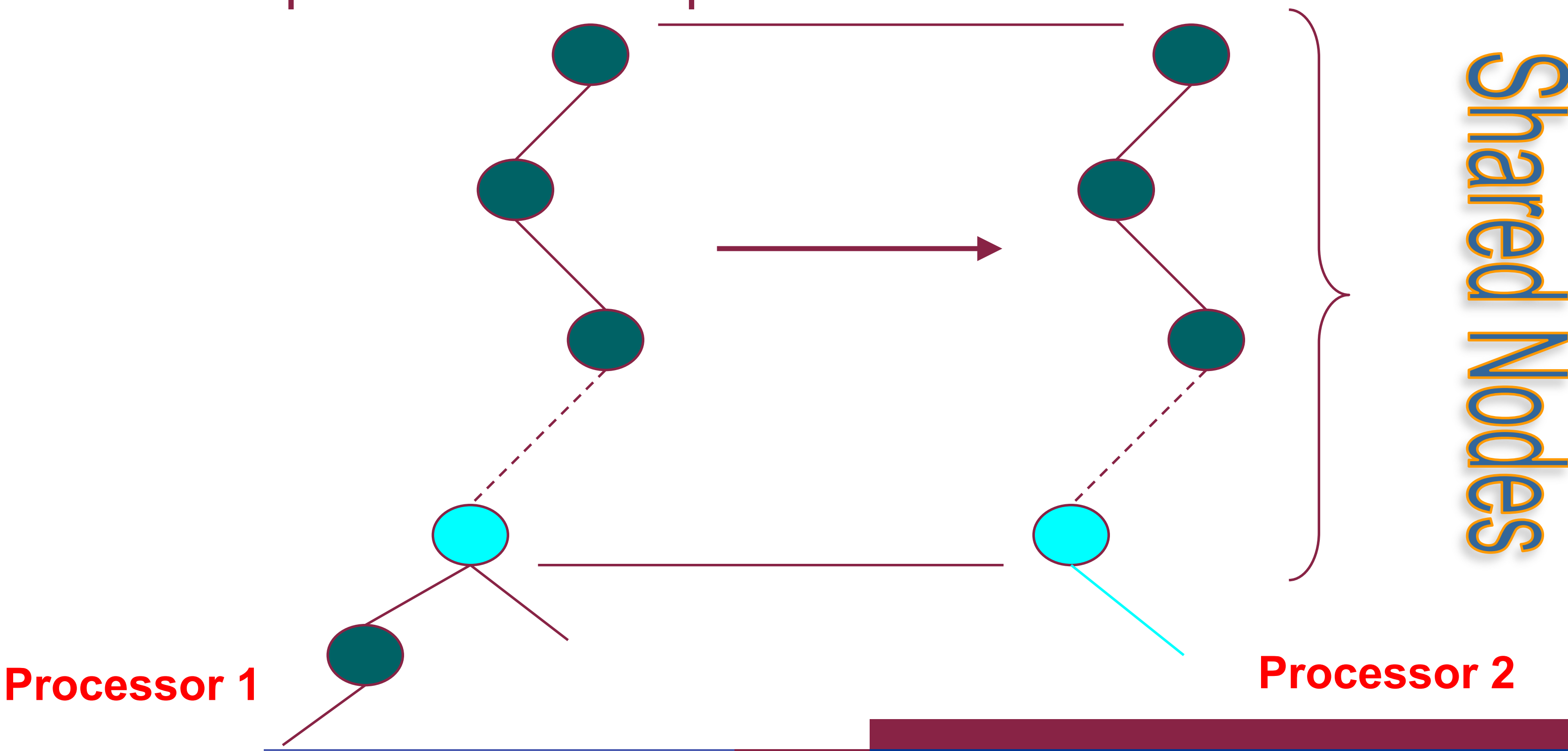
q(Y,Z) :- …Y=b, .. r(Z)…
q(Y,Z) :- ….

r(Z) :- … Z= …
r(Z) :- … Z=...



Stack Copying

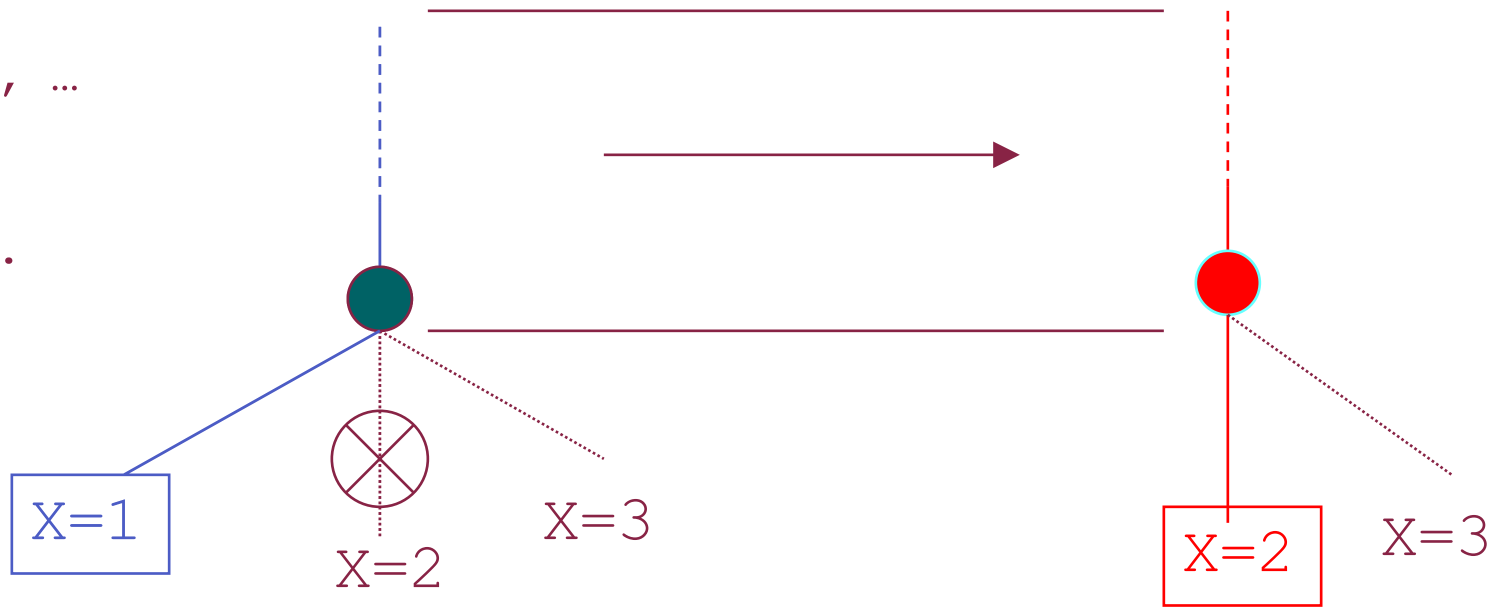
- Solve environment representation by duplicating the shared part of the computation tree



Stack Copying

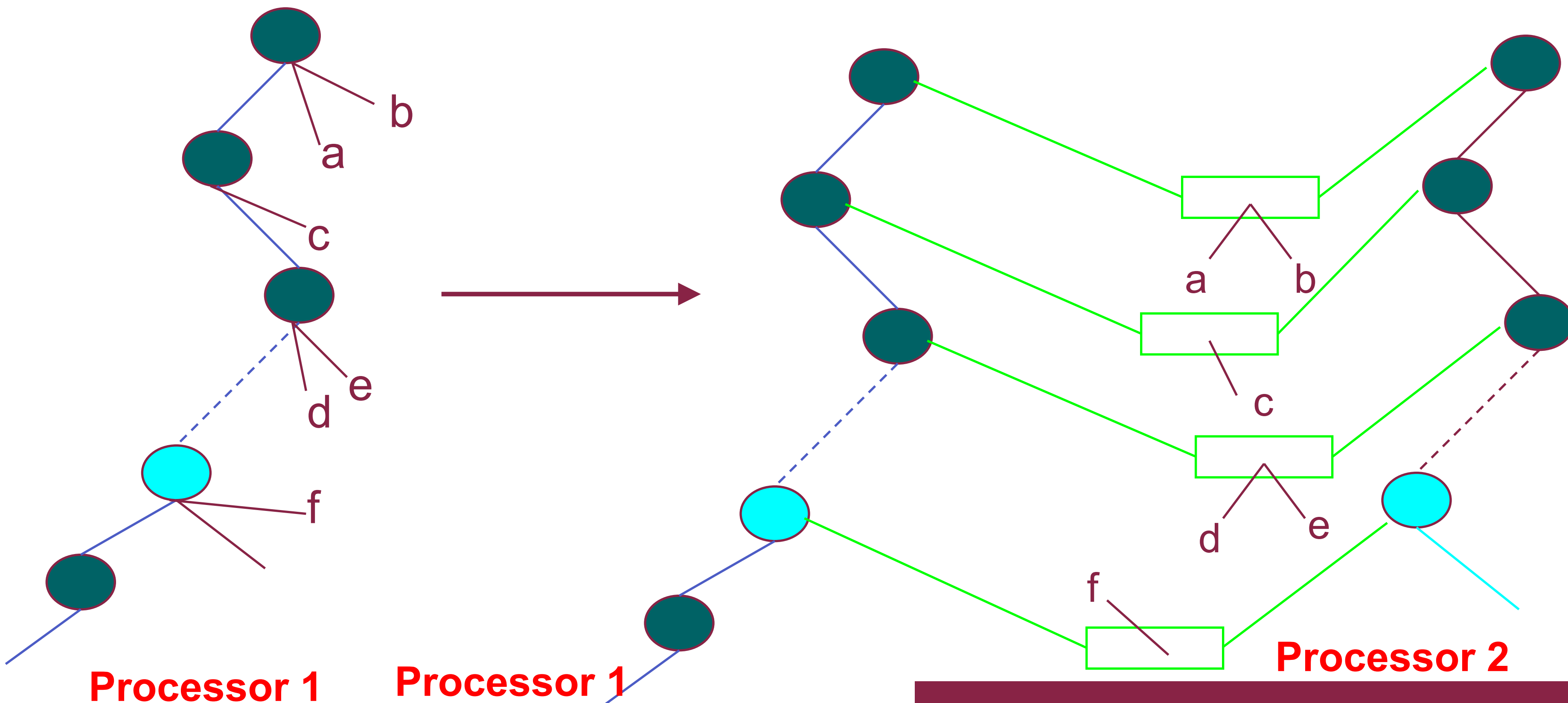
- Synchronization on shared choice points

```
?- …, p(X), …
p(1) :- …
p(2) :- …
p(3) :- ...
```



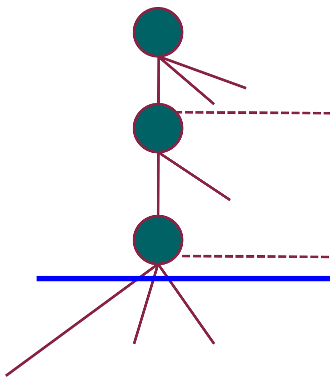
Stack Copying

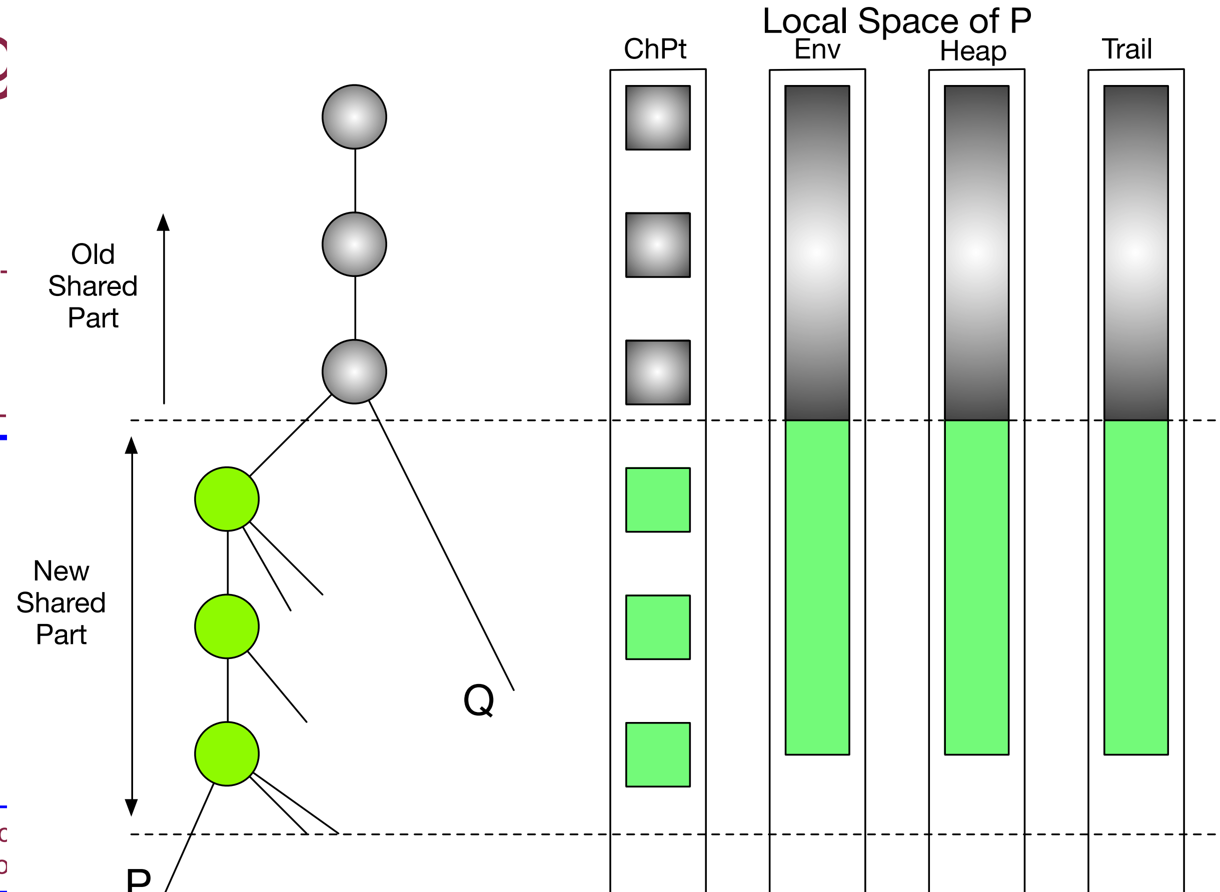
- Solution: make use of *Shared Frames*



Or-Parallelism: Binding Schemes

- Stack Copying





- Incrementality

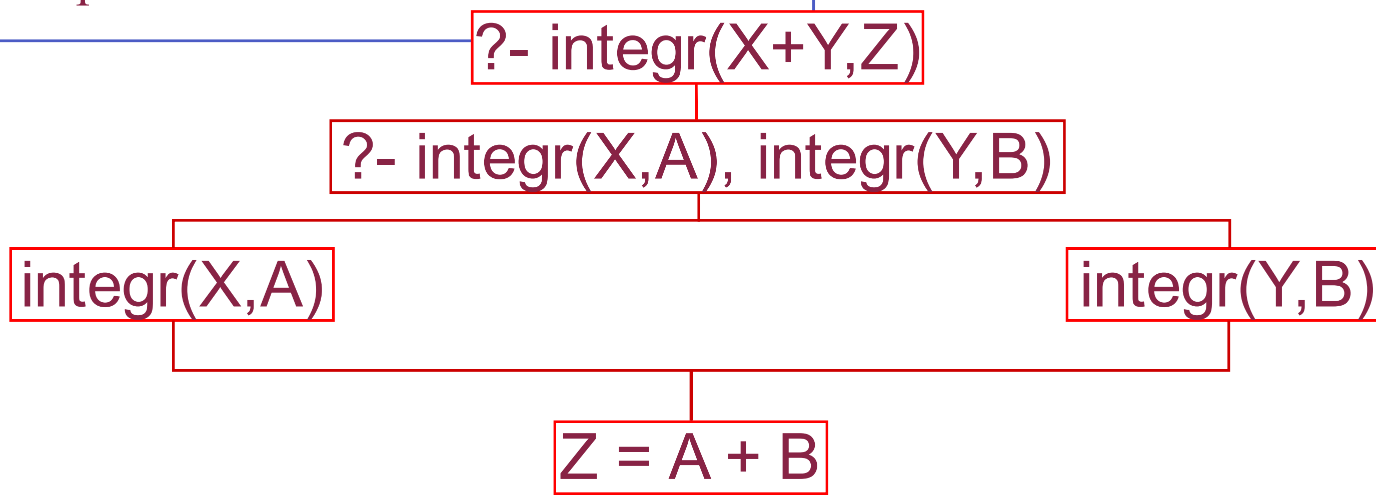
Ali, K. and Karlsson, R. 1990b. The MUSE Approac
International Journal of Parallel Pro

AND PARALLELISM

And-Parallelism

- Concurrent execution of different literals in a resolvent
- Mostly organized as fork-join structures

$$\text{integr(X + Y, Z) :- } \underbrace{\text{integr(X,A), integr(Y,B),}}_{\text{parallel literals}} \; \underbrace{\text{Z = A + B.}}_{\text{continuation}}$$



And-Parallelism

- Two traditional forms
  - Independent And-Parallelism
    - runtime access to independent sets of variables

parallel case

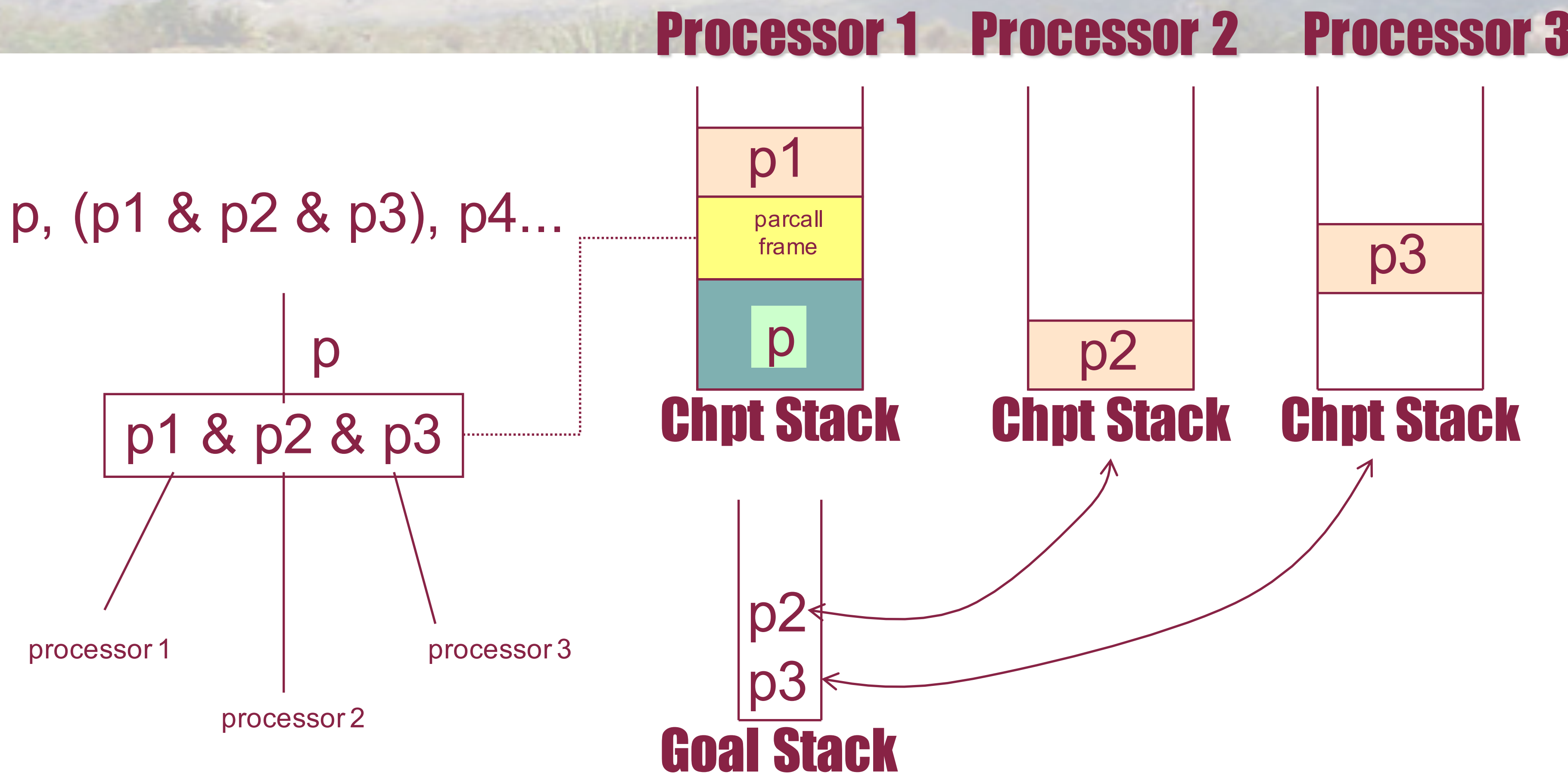
```
quick(In,Out) :- partition(In,First,Low,High),
                 ( indep(Low,High) => quick(Low,SLow) & quick(High,SHigh)
                   ;      quick(Low,SLow) , quick(High,SHigh)
                 ),
                 append(SLow,[First|SHigh],Out).
```

test

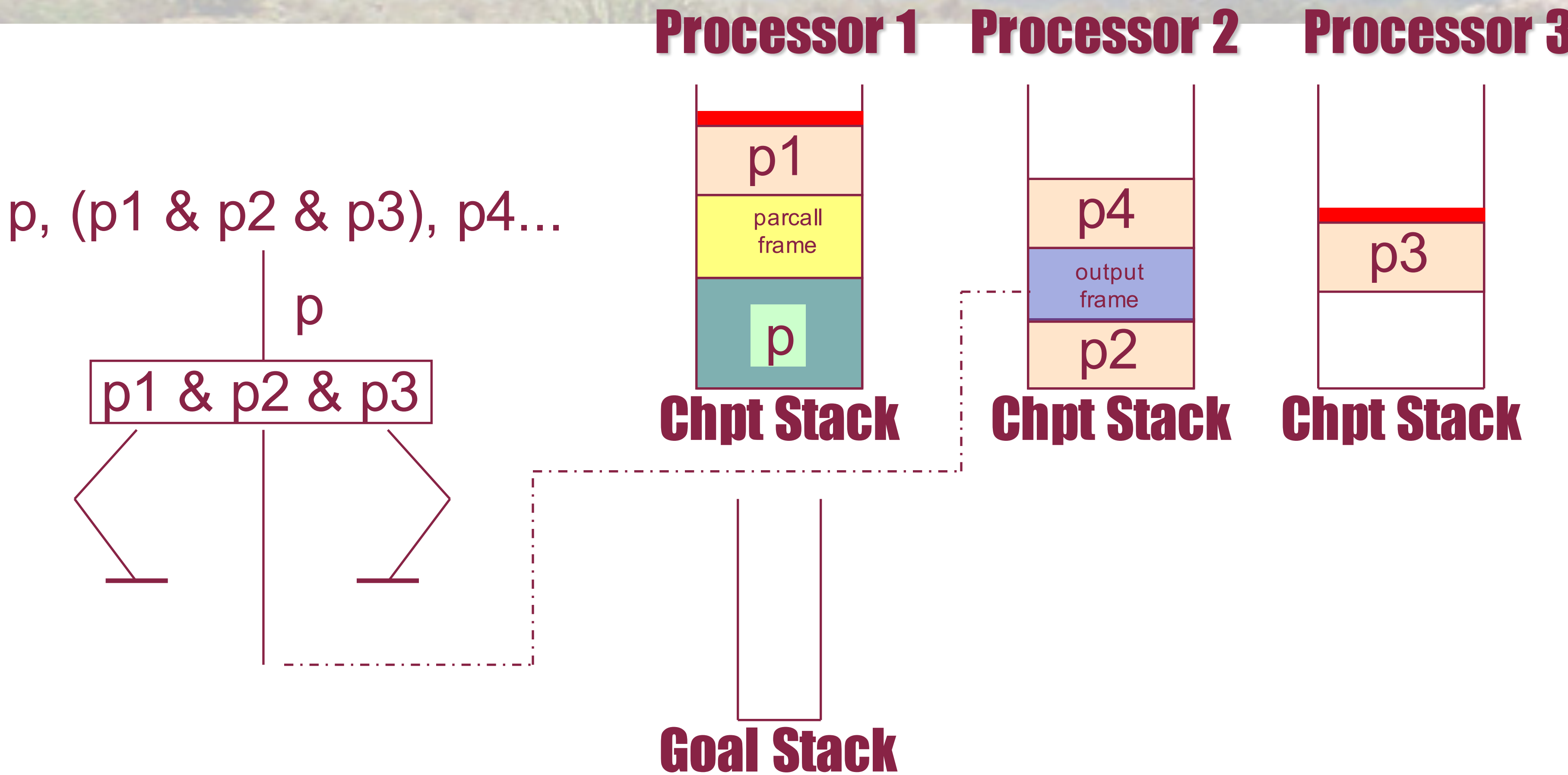
sequential case

Hermenegildo, M. 1986. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In Proceedings of the International Conference on Logic Programming, Springer-Verlag, 25–40.

And-Parallelism



And-Parallelism



And-Parallelism

- Backtracking

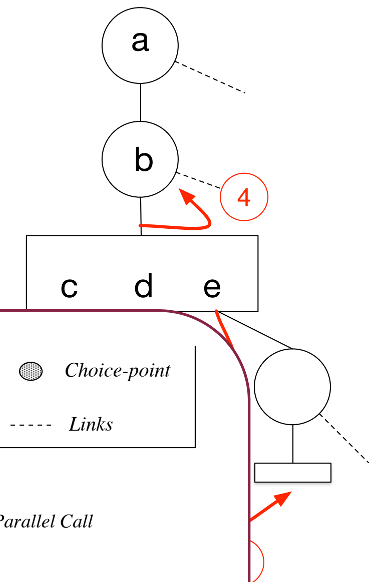
p1(..), (<cond> $\Rightarrow$ p2(..) & p3(..) & p4(..)), p5(..)

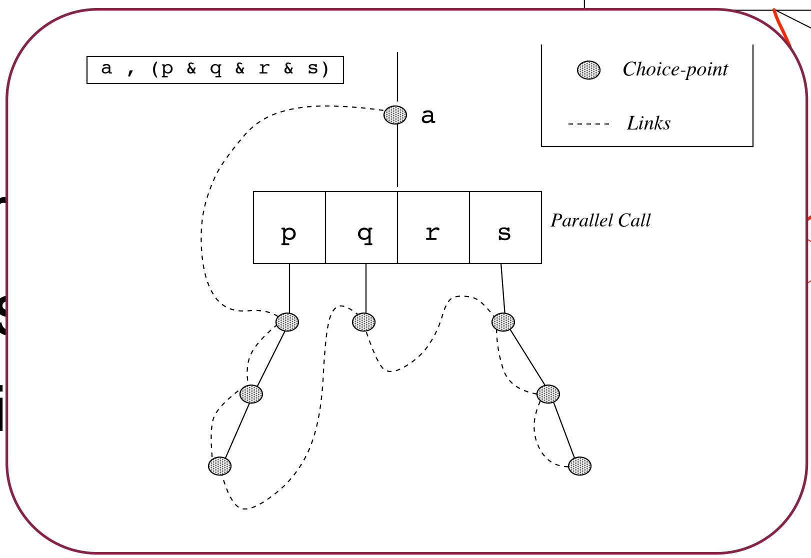
And-Parallelism

- Outside backtracking

  b(..), (<cond> ⇒ c(..) & d(..) & e(..)), ~~f(..)~~

- Standard right-to-left
  - across processors
    - Public vs Private Backtr
  - skip deterministic goals
    - Choice points linearizati
  - restart in parallel





And-Parallelism

- Inside backtracking

kill kill

p1(..), (<cond> ⇒ p2(..) & ~~p3(..)~~ & p4(..)), p5(..)

- Optimizations
  - Speculative Backtracking + Memoization

Pontelli, E. and Gupta, G. 2001, Backtracking in Independent And-Parallel Implementations of Logic Programming Languages. IEEE Trans. Parallel Distrib. Syst. 12(11): 1169-1189.

P. Chico de Guzmán, A. Casas, M. Carro, M.V. Hermenegildo:2011. Parallel backtracking with answer memoing for independent and-parallelism. TPLP 11(4-5): 555-574

And-Parallelism

- Dependent and-parallelism

  p(X) & q(X)

  - Goals
    - consistent bindings
    - reproduce Prolog observable behavior
  - Different Degrees of Dependence
    - non-conflicting: actually no conflict is present on shared variables (e.g., Non-strict Independence)
    - determinate: only determinate parallel subgoals (e.g., Basic Andorra Model)
    - unrestricted: no restrictions on parallel subgoals (e.g., DDAS, ACE)

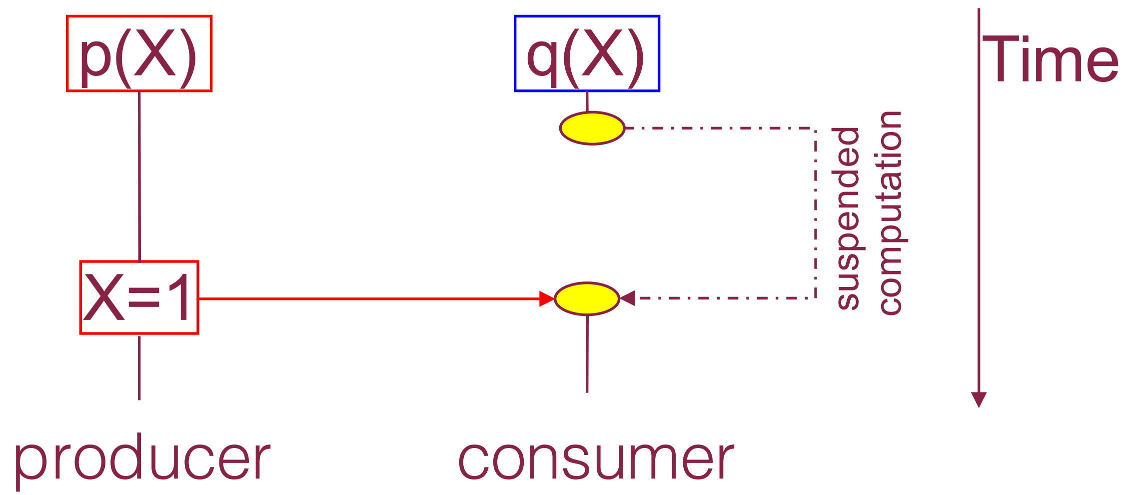
Shen, K. 1996. Overview of DASWAM: Exploitation of Dependent And-Parallelism. Journal of Logic Programming 29, 1/3, 245–293.
Pontelli, E. and Gupta, G. 1997. Implementation Mechanisms for Dependent And-Parallelism. In Proceedings of the International Conference on Logic Programming, L. Naish, Ed. MIT Press, Cambridge, MA, 123–137.

And-Parallelism

- Common approach
  - dynamic classification of subgoals as producers/consumers
    - Produces are allowed to bind variables
    - Consumers can only read bindings, not create them
  - several complex schemes (e.g., filtered binding model, DDAS)
- Complex backtracking

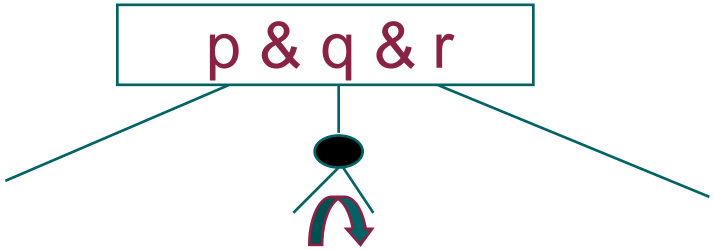
?- p(X) & q(X)

p(2).
p(1).
q(3).
q(2).



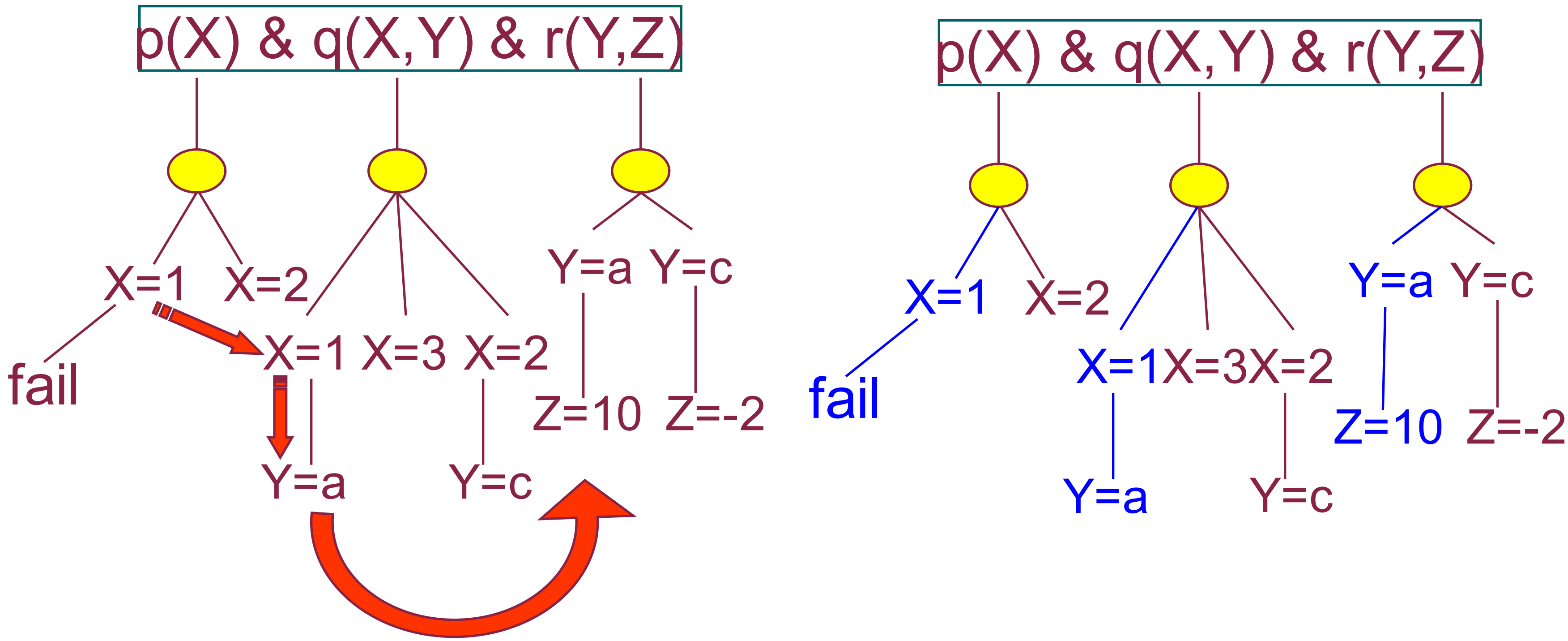
Backtracking

- Backward execution is complex!
  - Outside Backtracking: unchanged
  - Inside Backtracking: backtracking is not “independent”



- In indep. and-parallelism: backtracking in `q` will affect `p, r` only if `q` completely fails.
- In dependent and-parallelism even backtracking inside a subgoal may affect other subgoals - due to dependent variables

Backtracking



- propagation of backtracking across subgoals
  - a new notion of speculative work: dependent and-parallel computations which consume a *conditional* binding

Backtracking

- Extended Point-Backtracking
  - subgoals subdivided into groups;
  - if G_1 and G_2 are two groups, then $vars(G_1) \cap vars(G_2) = \emptyset$

$$\boxed{a(A)} \quad \& \quad \boxed{b(A)} \quad \& \quad \boxed{c(B)} \ \& \ \boxed{d(C)} \ \& \ \boxed{e(B)} \quad \& \quad \boxed{f(A)}$$

- groups are independent - if a group fails, then the whole parallel call fails
- inside a group:
  - if leftmost subgoal of group fails then the whole group fails (e.g., `c(B)`)
  - if non-leftmost subgoal of group fails then backtracking should continue in the immediately preceding subgoal in the group (e.g., from `b(A)` to `a(A)`)
  - if a subgoals untrails a dependent variable, then all the subgoals on its right in the group are restarted (e.g., `a(A)` changes the value of `A`, then `b(A)` and `f(A)` are restarted)

Committed-choice Languages

- "The Great Schism" [Warren'93]
- or-control: committed-choice $select_{clause}$ does not generate choice points
- and-control: data-flow dependent and-parallelism

$$\boxed{p(\ldots) \text{ :- } guard_1, \ldots, guard_n \mid goal_1, \ldots, goal_m}$$

- producer-consumer management of dependent and-parallelism
- producer/consumers detected through
  - modes (Parlog)
  - variable annotations (Concurrent Prolog)
  - implicit modes (GHC, KL1)
    - guards not allowed to bind external variables (suspends)

?- p(X), q(X).

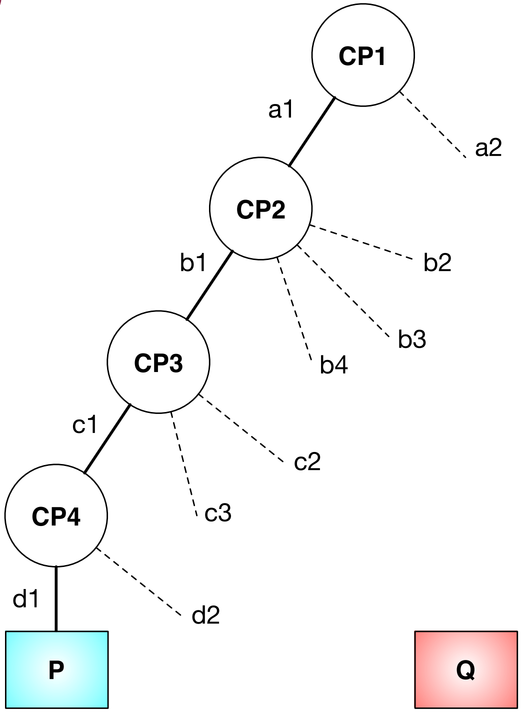
p(A) :- A=ok | ...

q(B) :- true | B = ok.

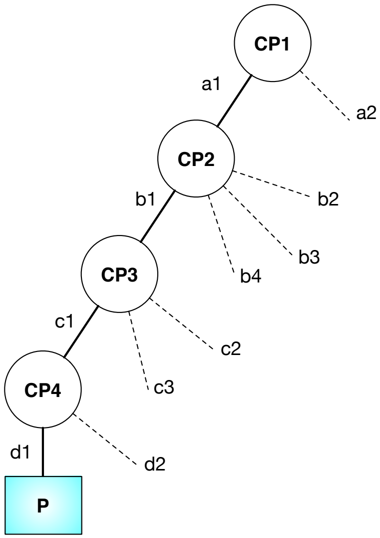
MORE RECENT APPROACHES

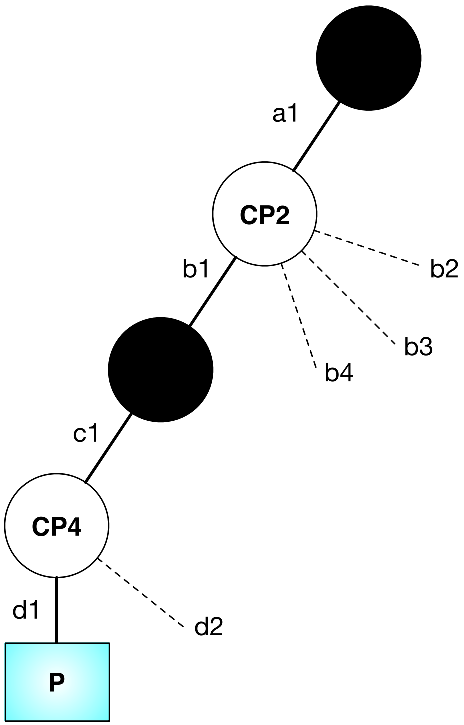
More Recent Techniques

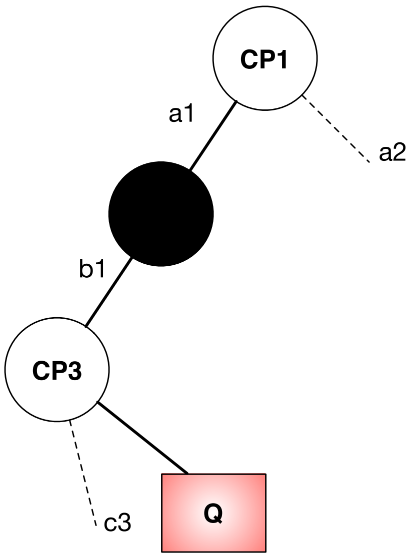
- Or-Parallelism via Stack Splitting
- Copy nodes from P to Q
  - Incremental Copying:
    - From bottommost open node of P to bottommost node in common with Q
  - (on shared memory):
    - Create shared frames for copied nodes



Vertical Stack Splitting

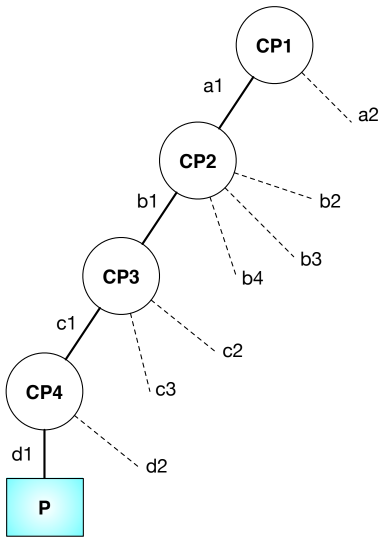


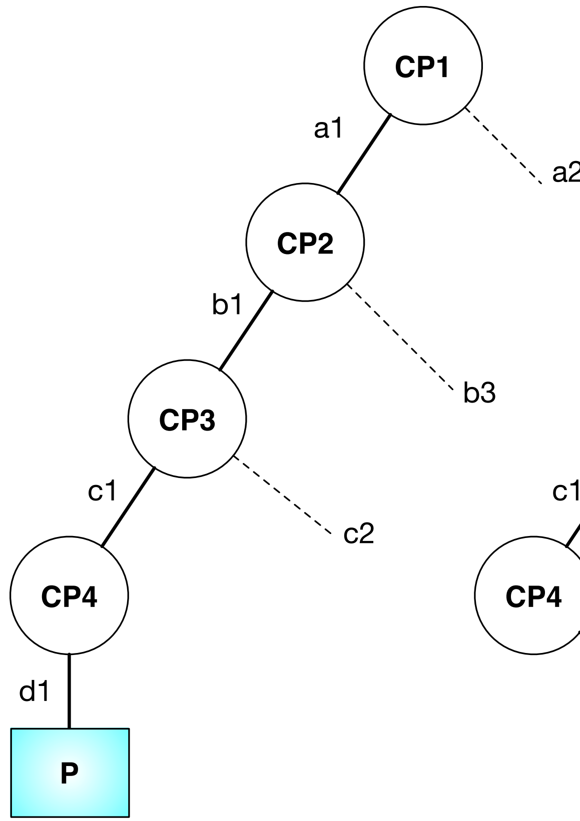


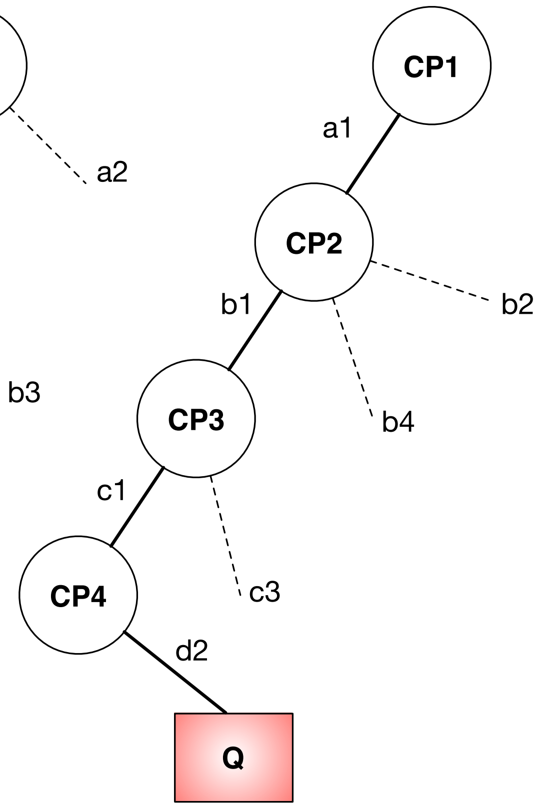


E. Pontelli, K. Villaverde, H-F. Guo, G.Gupta. 2006. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. J. Parallel Distrib. Comput. 66(10): 1267-1293.

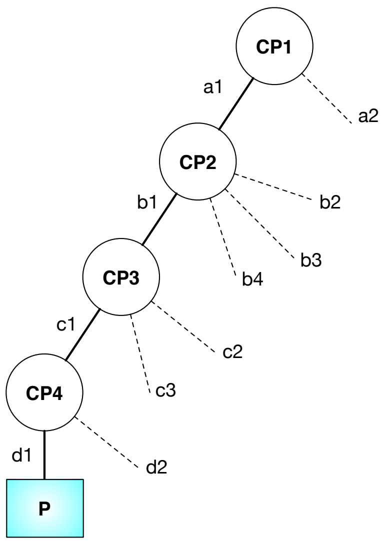
Horizontal Stack Splitting

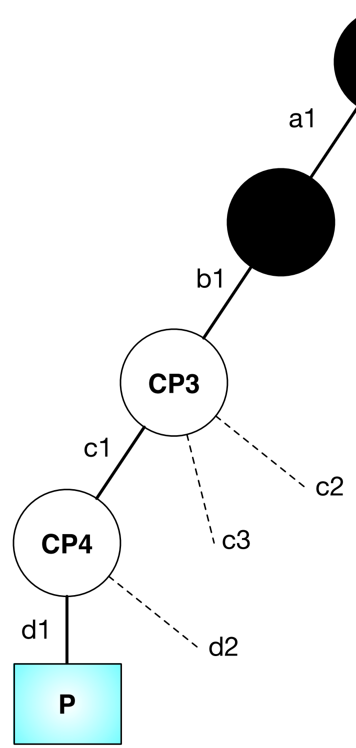


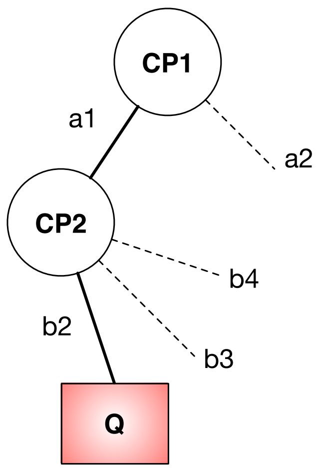




Half Stack Splitting



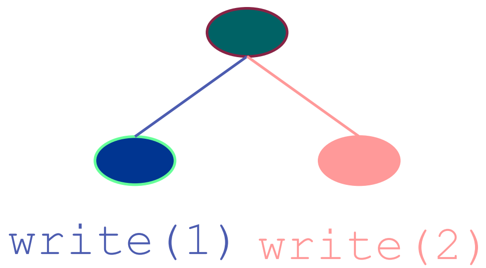




R. Vieira, R. Rocha, R.M. A. Silva, 2013. On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. CoRR abs/1301.7690.

Side-Effects and order-sensitive predicates

- Side-effects are order-sensitive predicates: their behavior depends on the order of execution
- E.g.,



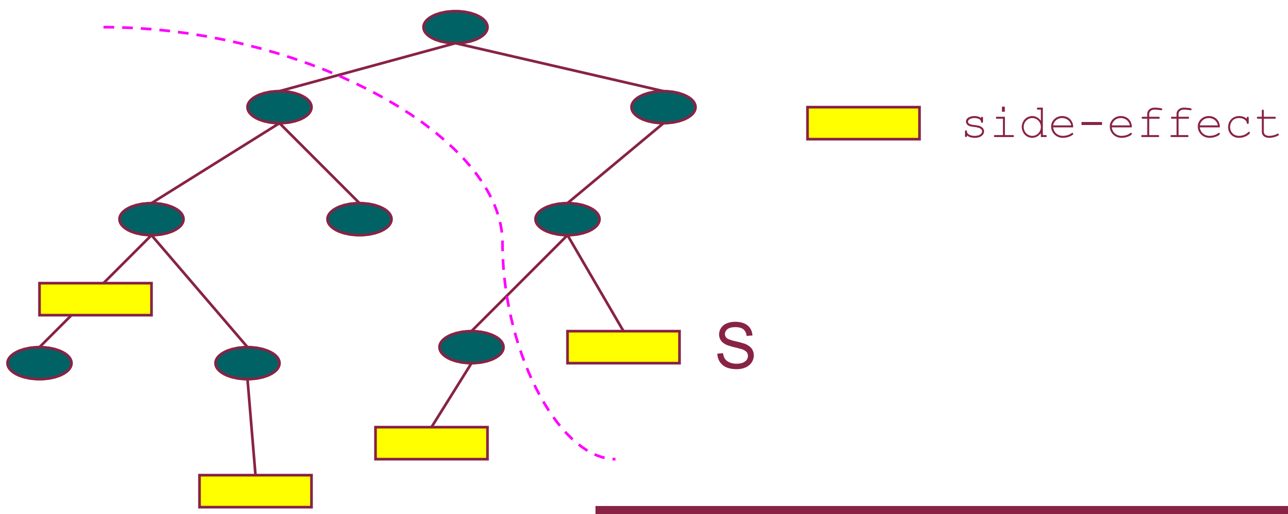
2
1

- Goal: recreate the same observable behavior of sequential Prolog
- Sequentialize order-sensitive predicates
- Sequential is opposite of Parallel…
- Dynamic vs. Static management of order-sensitive predicates

K. Villaverde, E. Pontelli, H-F. Guo, G. Gupta, 2003. A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. Euro-Par, Springer Verlag, 694-703.

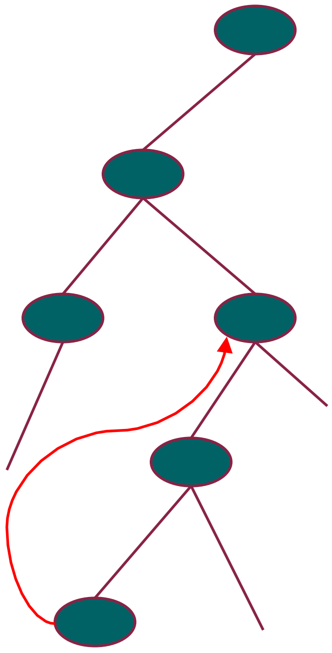
Order-sensitive Executions

- Idea: a side-effect should be delayed until all “preceding” side-effects have been completed
- determining the exact time of execution: undecidable
- safe approximation: delay until all “impure” branches on the left of the side-effects have been completed



Order-sensitive predicates

- Standard Technique: maintain subroot nodes for each node
- Subroot(X) = root of largest subtree containing X in which X is leftmost



- Aurora, Muse: O(n) algorithms for maintaining subroot nodes
- possible to perform O(1)

Tutorial Roadmap

Basics

Prolog

YESTERDAY

Systems

Ciao clasp

YAProlog

ASP

today

Going Large

Going Small

(EARLY) TOMORROW

Where are the systems?

To Japanese Page for "The Home page of KLIC"

This page describes KLIC, which is a concurrent logic programming language.

KLIC was developed in ICOT and distributed as an IFS, and is now distributed by KLIC Association.

Please read the README file for details

- Current version
- Available platforms
- Parallel implementations
- Changes
- Documents
- Mailing lists

Current version

Version 3.003 is the latest and is currently distributed from the following sites:

- http://www.klic.org/files/v3.0/klic-3.003.tgz(primary site)
- ftp://pwd.chroot.org/pub/klic/v3.0/klic-3.003.tgz(secondary site)

This distribution contains the following three implementations.

- Sequential (pseudo parallel) implementation
- Distributed memory parallel implementation
- Shared memory parallel implementation

Unfortunately, the current shared memory parallel implementation still has fatal bugs. If you are using the old shared memory parallel implementation (previous version is 2.002), *please do not replace it with the current version.*

HTTP Status 404 - /~beaumont/andorra.html

type Status report

message /~beaumont/andorra.html

description The requested resource (/~beaumont/andorra.html) is not available.

Apache Tomcat/5.5.9

HTTP Status 404 - /~beaumont/aurora.html

type Status report

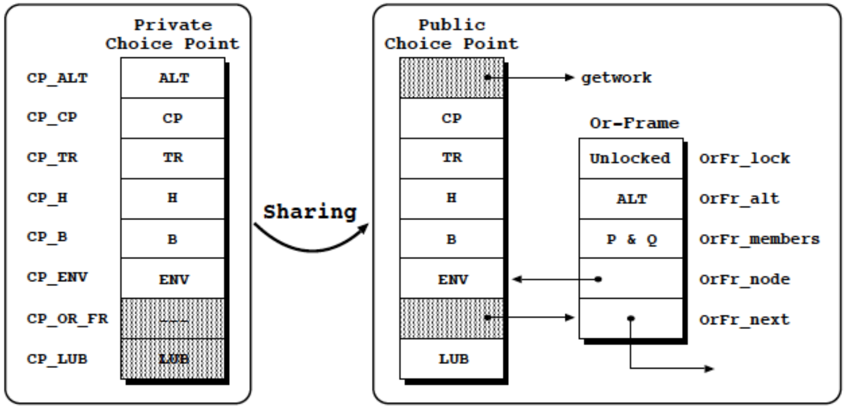
message /~beaumont/aurora.html

description The requested resource (/~beaumont/aurora.html) is not available.

Apache Tomcat/5.5.9

YAP: Perfect Engineering of Or-P

- Supports Or-Parallelism through splitting
- Different engines
  - YapOr:
    - Shared memory system, workers using processes
    - Supports both or-frames model and stack splitting
  - ThOr:
    - Shared memory system, workers as threads
      - More complex copying – require shifting pointers
    - Easier to port on different platforms
    - Restricted to pure Prolog
  - Teams:
    - Team = process
    - Team members = threads within the process
    - Designed to run on multiprocessors of multicores



J. Santos, R. Rocha, 2013. Or-Parallel Prolog Execution on Clusters of Multicores. SLATE, 9-20

CIAO Prolog: Making Parallel Prolog Boring

- High level parallel primitives
  - Implement forms of parallelism in Prolog
  - Original idea
    - Codish & Shapiro (1987)
    - &-Prolog CGE (1988)
    - &-ACE DAP (1995)
- Raise implementation from abstract machine/compiler to source code
- Experimented with at the level of IAP

A. Casas, M. Carro, M.V. Hermenegildo, 2008. A High-Level Implementation of Non-deterministic, Unrestricted, Independent And-Parallelism. ICLP, Springer Verlag, 651-666

CIAO Prolog

- Source-level operators
  - `G &> H`
    - G scheduled for parallel execution
    - Thread places goal on its goal queue
    - H handler of G
  - `H <&`
    - Waits for goal with handler H to terminate
    - Upon termination, all bindings of G are available
- Traditional & operator is now:

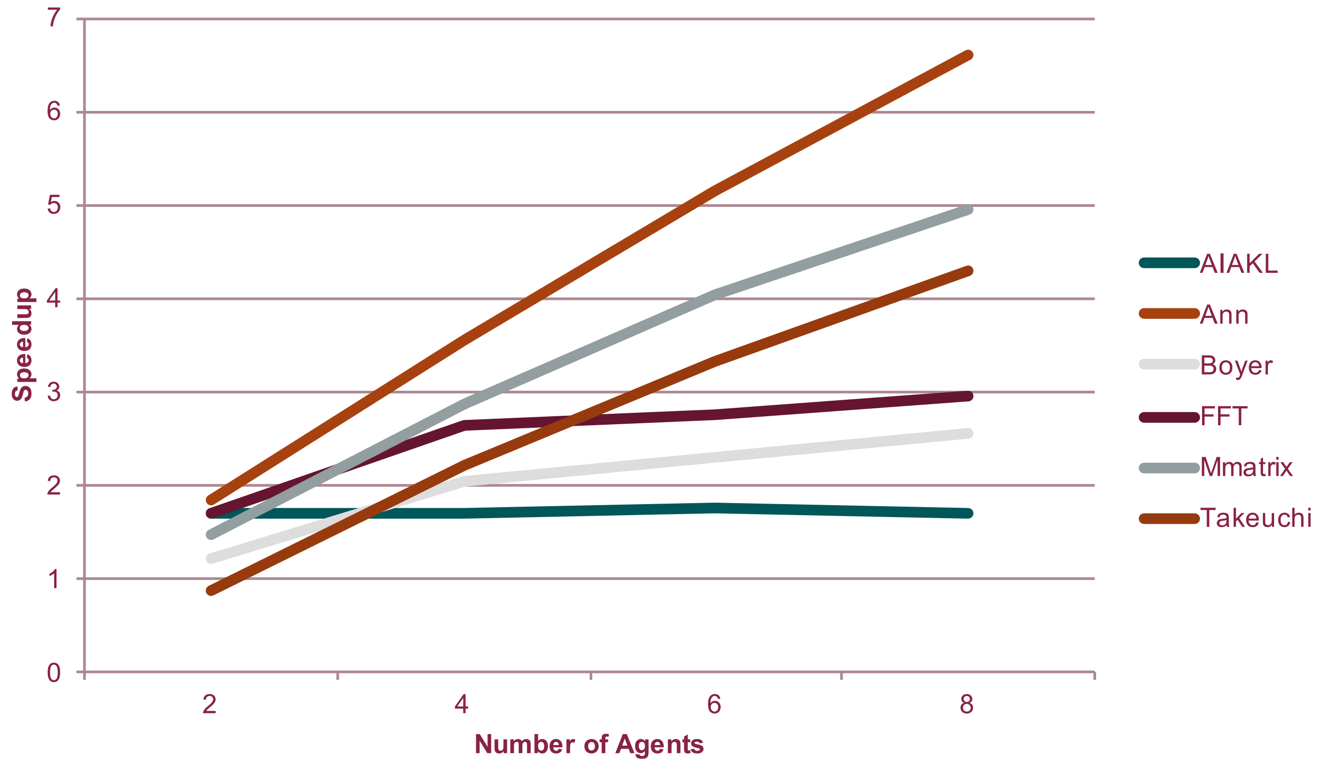
```
A & B :- A &> H, call(B), H<&.
```

Implementation

- Sample code fragment

Handler

Handler



Tutorial Roadmap

Basics

Prolog

YESTERDAY

Systems

Ciao

clasp

YAProlog

ASP

TODAY

Going Large

Going Small

(EARLY) TOMORROW

Parallelism in ASP

- Implementations based on search techniques
- Similar Foundations as Prolog
- First proposals in 2001
  - Finkel et al. 2001 (parstab)
  - Pontelli et al. 2001

E. Pontelli, O. El-Khatib: Exploiting Vertical Parallelism from Answer Set Programs. Answer Set Programming, AAAI Spring Symp., 2001
R. Finkel, V.Marek, N. Moore, M. Truszczynski: Computing stable models in parallel. Answer Set Programming, AAAI Spring Symp., 2001

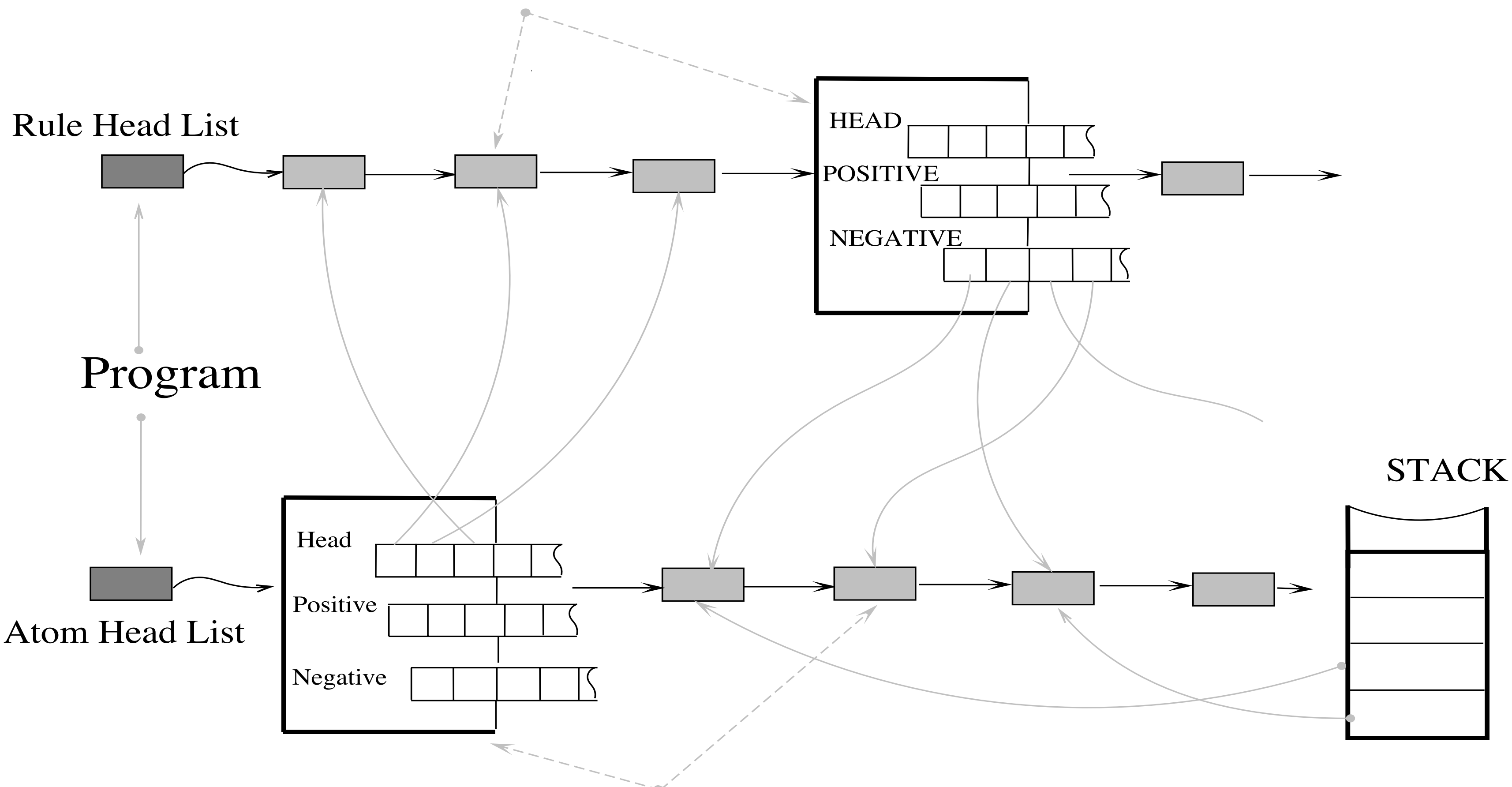
Basic Procedure

- Both systems use Smodels as their core

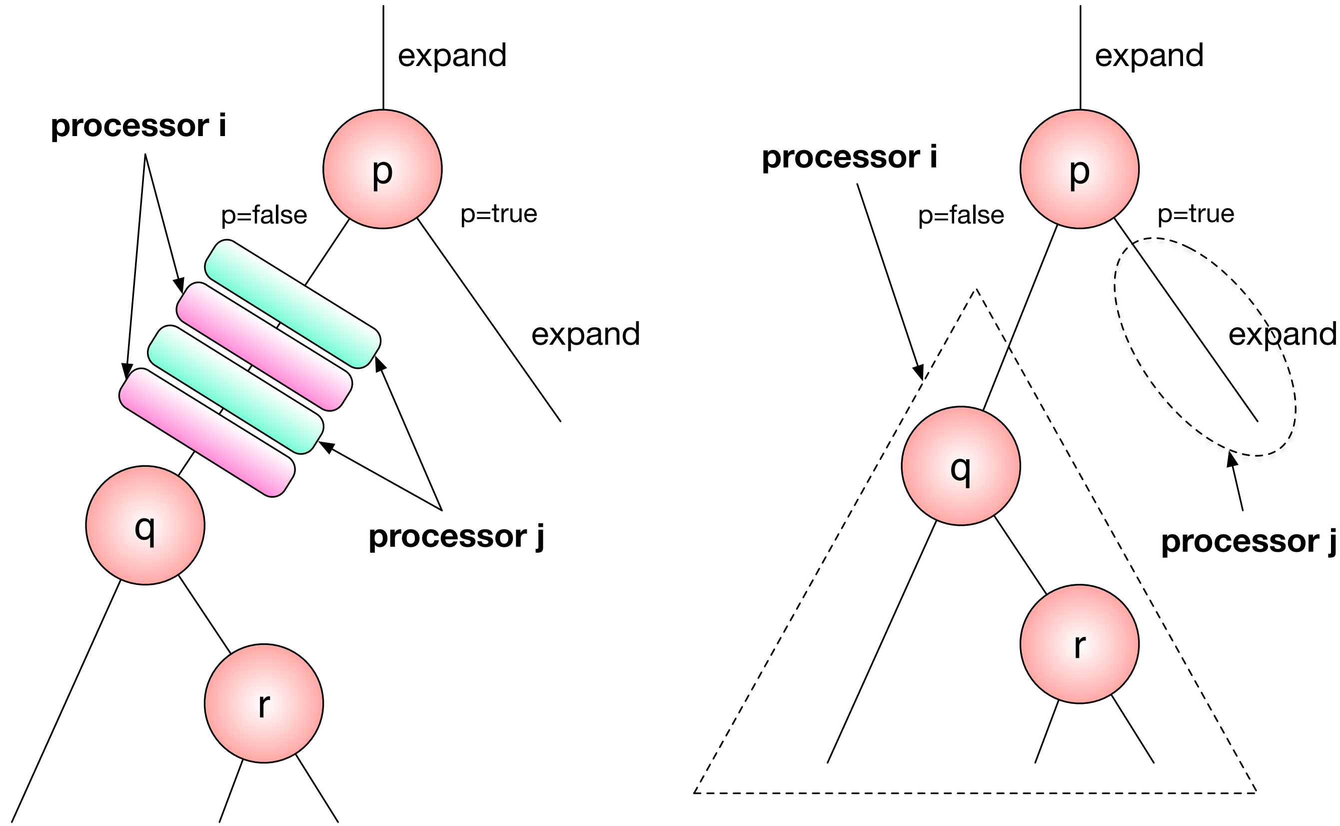
Compute (P: Program)
$S := \langle \varnothing, \varnothing \rangle$
while (TRUE) **do**
  S := EXPAND(P, S)
  if ($S^+ \cap S^- \neq \varnothing$) **then**
    return FAILURE
  endif
  if ($S^+ \cup S^- = B_\Sigma$) **then**
    return S
  endif
  choose either
    $S^+ := S^+ \cup \{CHOOSE_P(S)\}$ **or**
    $S^- := S^- \cup \{CHOOSE_P(S)\}$
endwhile

- Expand(Program, Partial Answer Set):
  - Fixpoint computation
  - Well-founded model that expands a partial interpretation
- $CHOOSE_P$(Partial Answer Set):
  - Heuristic-based
  - Other techniques (e.g., Lookahead)

Basic Procedure



Forms of Parallelism



Horizontal Parallelism **Vertical Parallelism**

Horizontal Parallelism

- Generic parallelization of the expand operation attempted with very modest results
- In general, hard to produce great results
  - Theoretical limitations:
    - Analogous to unit propagation/arc consistency
    - Problem is log-space complete in P
    - [Kasif 90] unlikely there is a polylog time parallel algorithm (using polynomial resources)
  - Practical limitations: risk of highly sequential cases

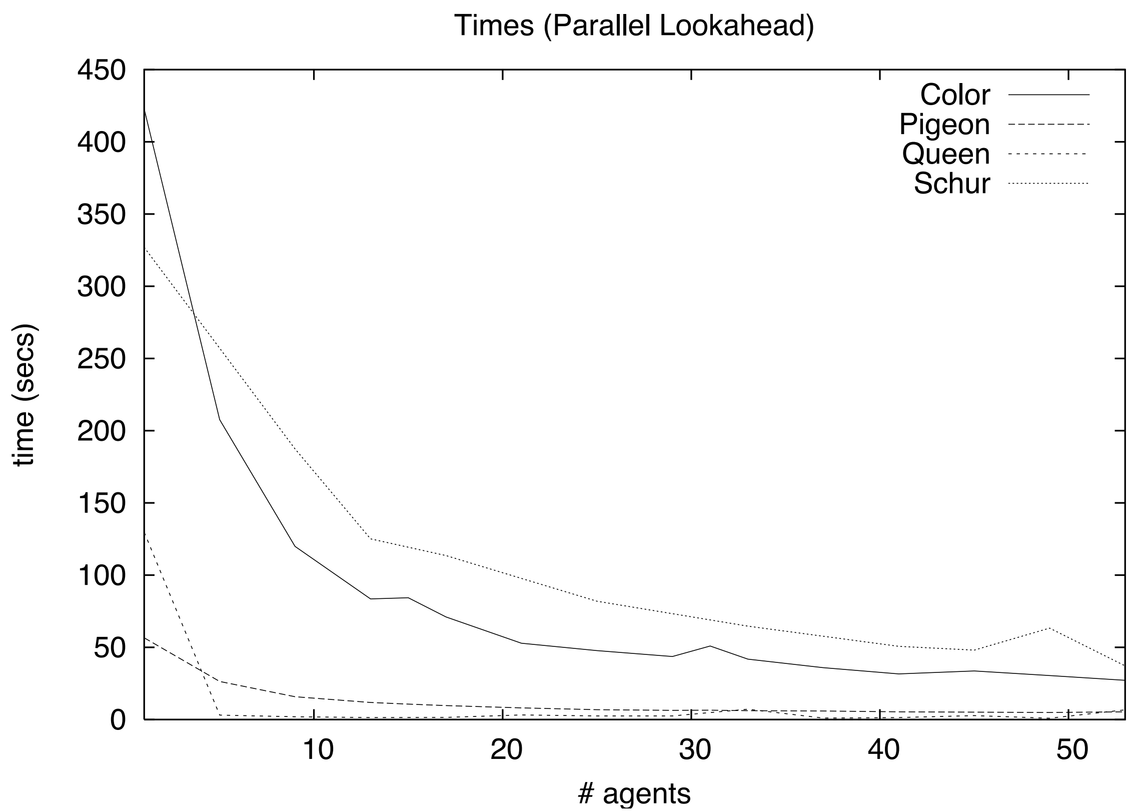
$q_1.$ p_1 :- $q_1.$ q_2 :- $p_1.$ p_2 :- $q_1.$ …

Horizontal Parallelism

- Parallel Lookahead
  - Before selecting a chosen atom
  - Test each undefined atom A:
    EXPAND($S \cup \{A\}$) and EXPAND($S \cup \{\text{not } A\}$)
    - If one leads to contradiction: deterministically add the complement
    - If both lead to contradiction: backtrack
    - Deterministic expansions; aid with heuristic
  - Perform test of each atom A in parallel

M. Balduccini, E. Pontelli, O. El-Khatib, H. Le, 2005. Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing 31(6): 608-647.

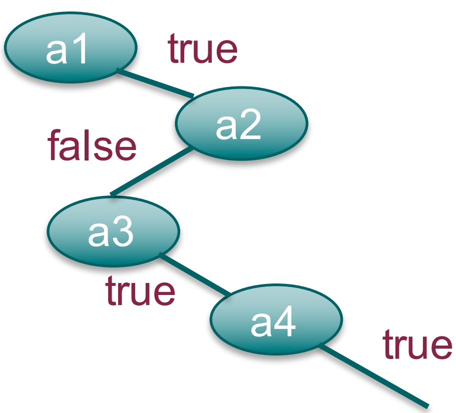
Horizontal Parallelism

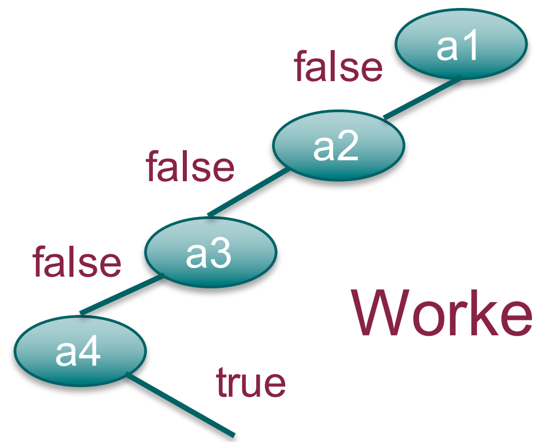


Design of a Vertical Parallel Engine

- Explore search (or-) parallelism in ASP
- Initial phase
  - Each worker receives complete ASP program
  - Each worker receives a unique binary ID (e.g., 1011)
  - Each worker uses ID to deterministically choose the first branches

Worker: **1011**

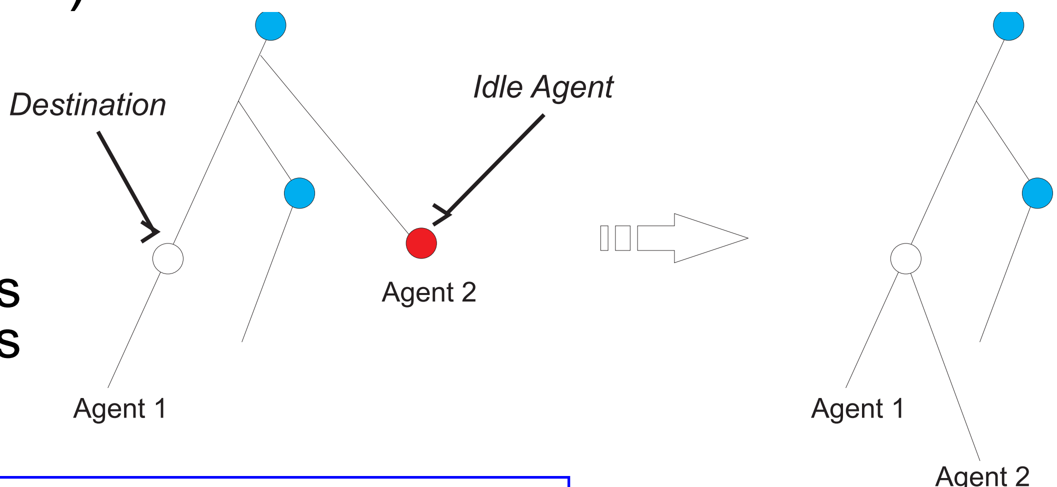




Worker: **0001**

Design of a Vertical Parallel Engine

- Symmetrical workers
- Each worker alternates
  - *Computation*: explore an assigned branch of the search tree (*local task*)
  - *Load Balancing*:
    - Idle worker moves to a different place in the search tree
    - Busy worker donates unexplored branches to idle worker



E. Pontelli, H. Le, T. Son, 2010. An investigation in parallel execution of answer set programs on distributed memory platforms. Computer Languages, Systems & Structures 36(2): 158-202.

Design of a Vertical Parallel Engine

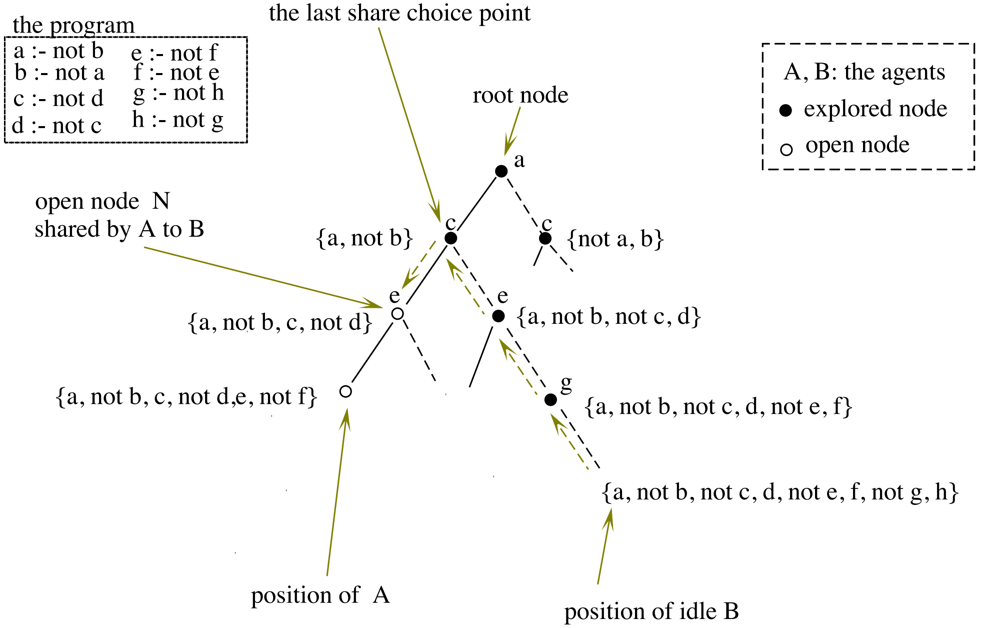
- Load Balancing composed of two activities
  - *Scheduling*: Identify the new position for a worker in the search tree
    - Worker from whom we are taking a choice point (***sender***)
    - Choice point that we are taking from such worker (*open node*)
  - *Task Sharing*: Position the idle worker to its new position

Task Sharing Strategies

- Two main categories
  - *Recomputation-based:* Receiver repeats computation of the sender to reach the open node
  - *Copying-based:* Sender provides copies of its data structure to allow receiver to directly jump to open node

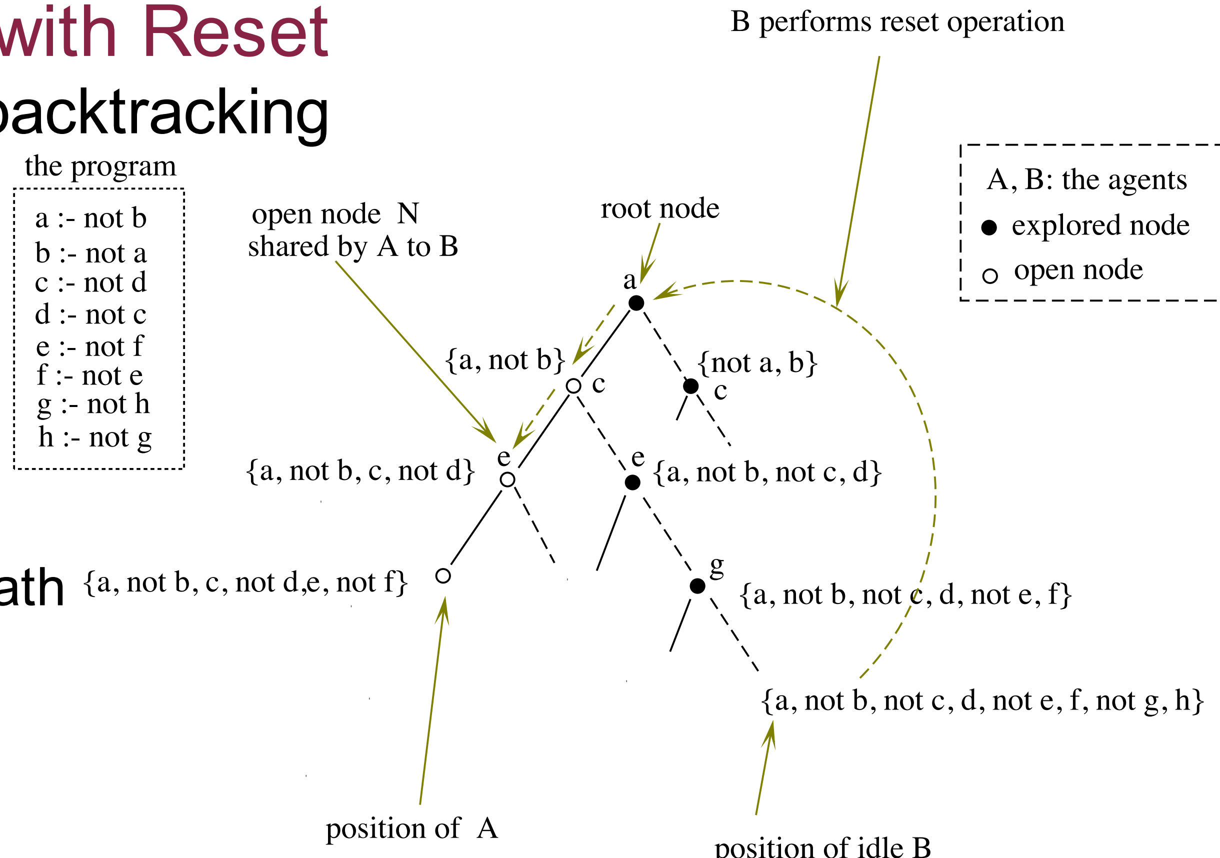
Task Sharing Strategies

- Recomputation with backtracking
  - Requires relative positions of workers for NCA computation
  - Exchange guiding paths
  - Variant:
    - Estimate NCA using broadcasts and half splitting



Task Sharing Strategies

- Recomputation with Reset
  - Avoid lengthy backtracking
  - No need for NCA
  - Only need guiding path
  - Variant:
    - Stack splitting for complete path of A

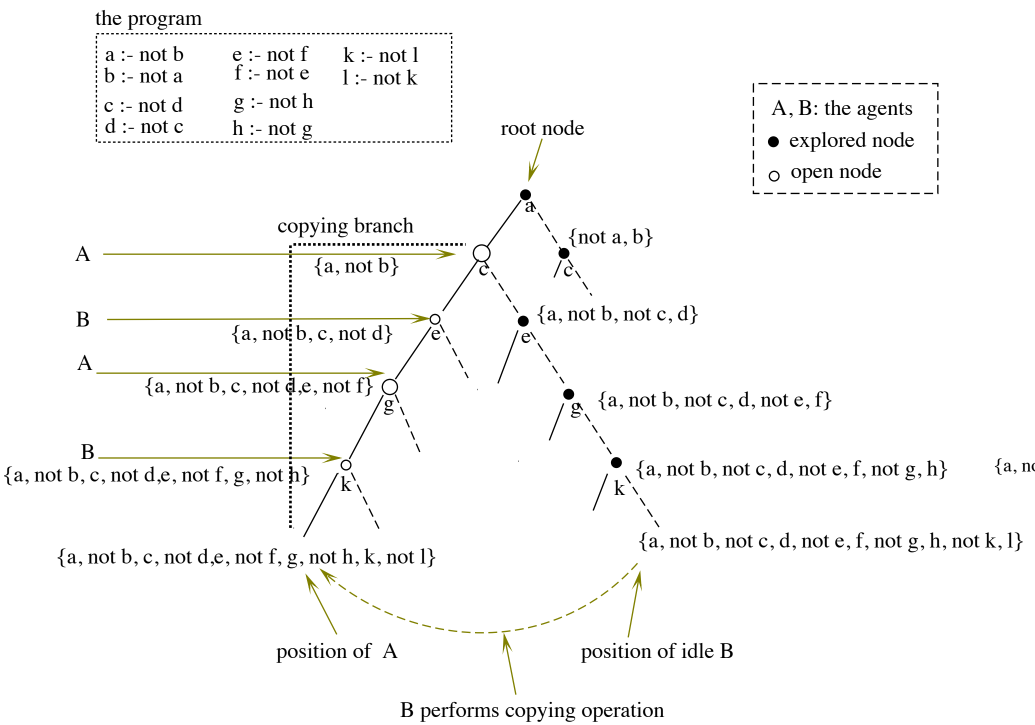


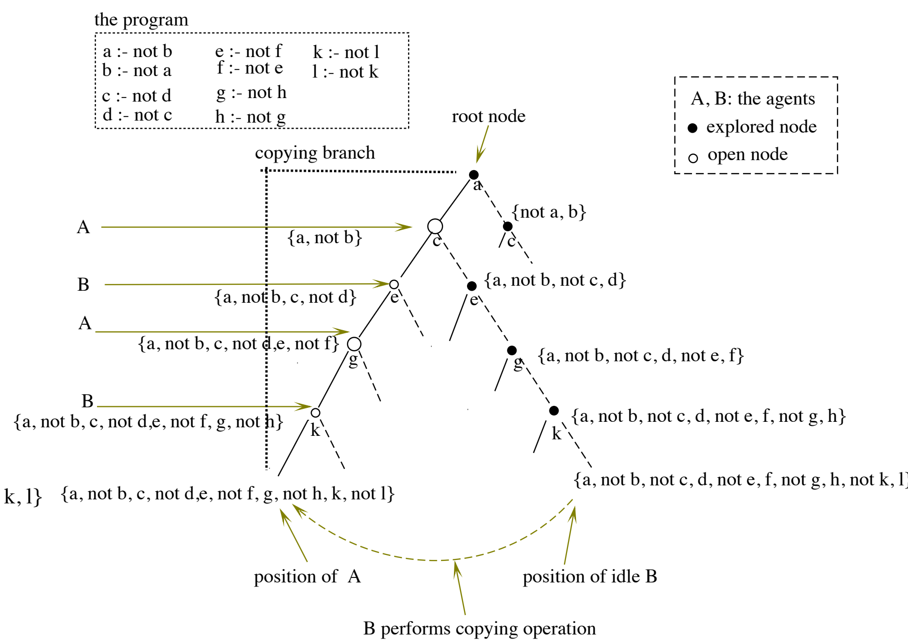
Task Sharing Strategies

- Copying
  - Sender gives copy of its data structures to receiver
    - State of rules and atoms
    - Atoms Stack
- Bottom-up approach
  - Copy to the current position of sender
  - Backtrack from there to open node
  - Natural to use stack splitting techniques to share multiple nodes at once

Task Sharing Strategies

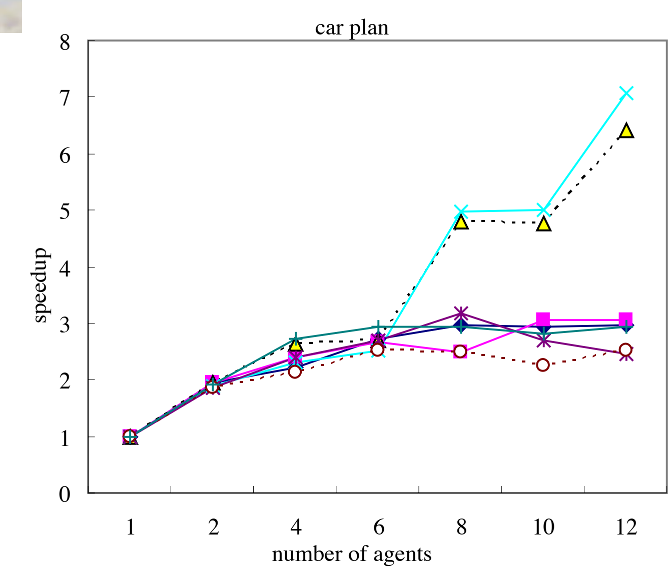
- Incremental Copying
- Copy-All

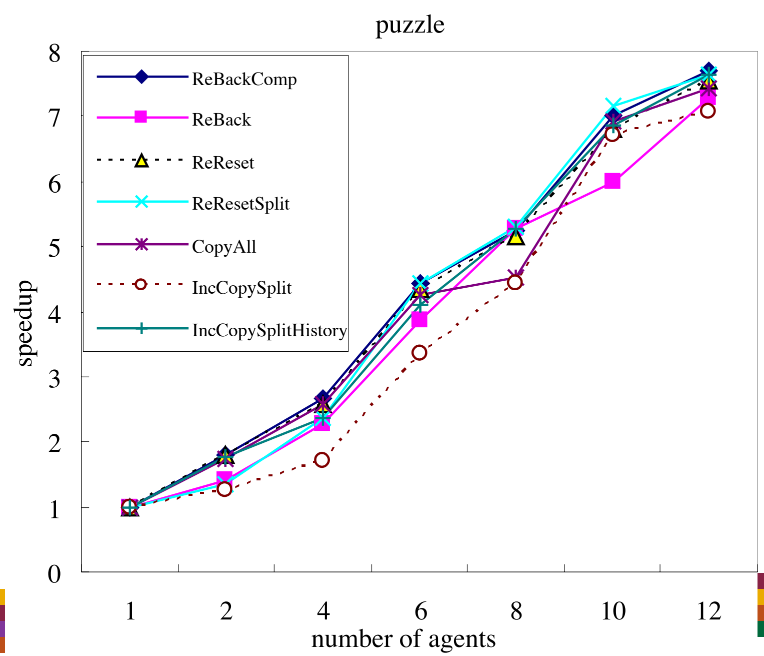


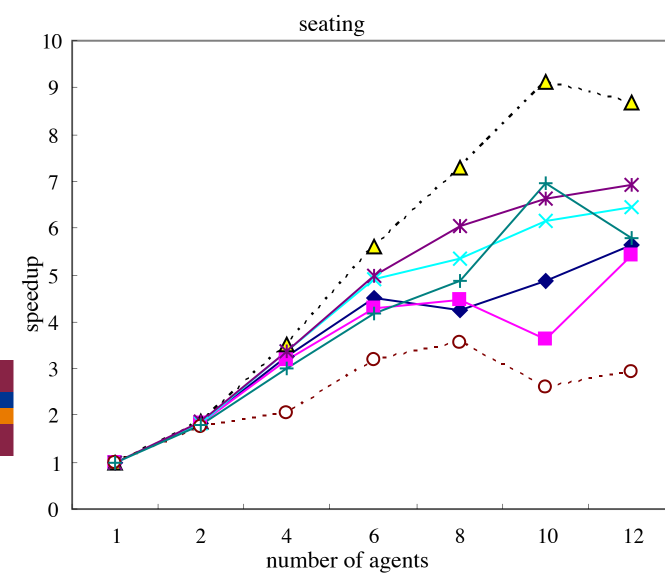


Task Sharing Strategies

- No clear winner!
- 60% variance in some cases



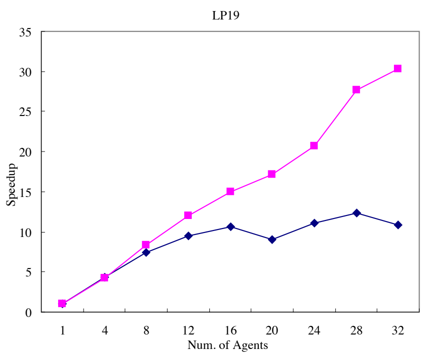


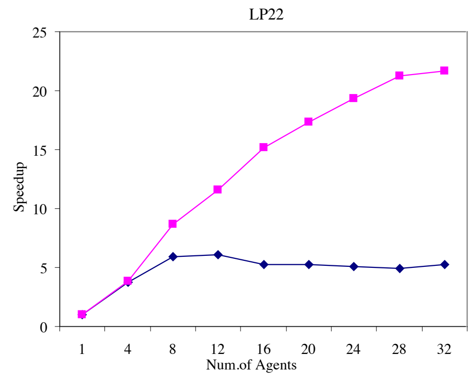


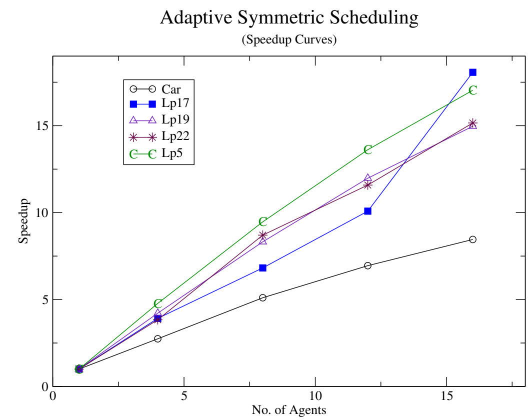
Scheduling

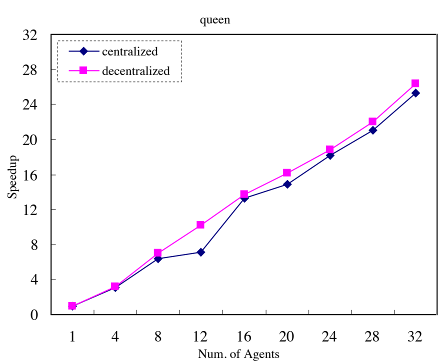
- Which workers will be involved
- Dimensions
  - Centralized vs Decentralized scheduling
  - Worker selection
    - Random
    - Based on local load
    - Based on location
  - Who initiates scheduling
    - Sender initiated vs. Receiver initiated

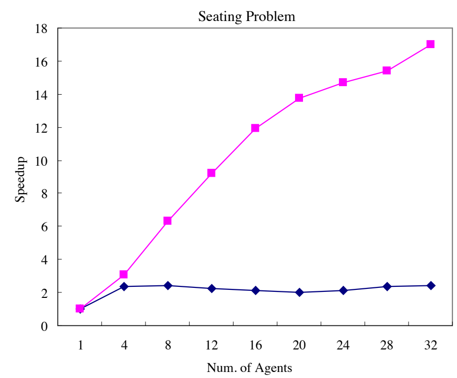
Centralized vs. Decentralized

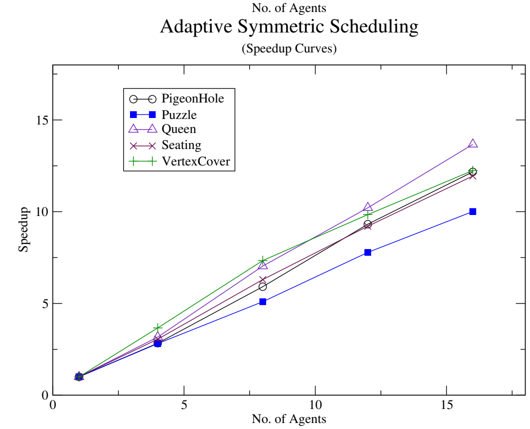












Production Systems: **clasp**

- Several multi-threaded versions
  - claspar
  - clasp 2
- Two approaches to parallelism
  - Vertical parallelism
    - Centralized Scheduling – queue of guiding paths
    - Dynamic load balancing
    - Up to 64 threads
  - **Portfolio parallelism**
- Advantage of threads
  - Easier to communicate
  - E.g., exchange of learned nogoods
    - Short nogoods only
    - Various filters to decide which nogoods to accept (e.g., how relevant to the current computation; how long; how many decision levels of literals)

Martin Gebser, Benjamin Kaufmann, Torsten Schaub, 2012. Multi-threaded ASP solving with clasp. TPLP 12(4-5): 525-545

Production Systems: **DLV**

- Two core forms of parallelism
  - Portfolio Parallelism
    - Each thread solves the problem with a different branching heuristic
    - Stop as soon as a thread finds a model
  - Parallel Grounding

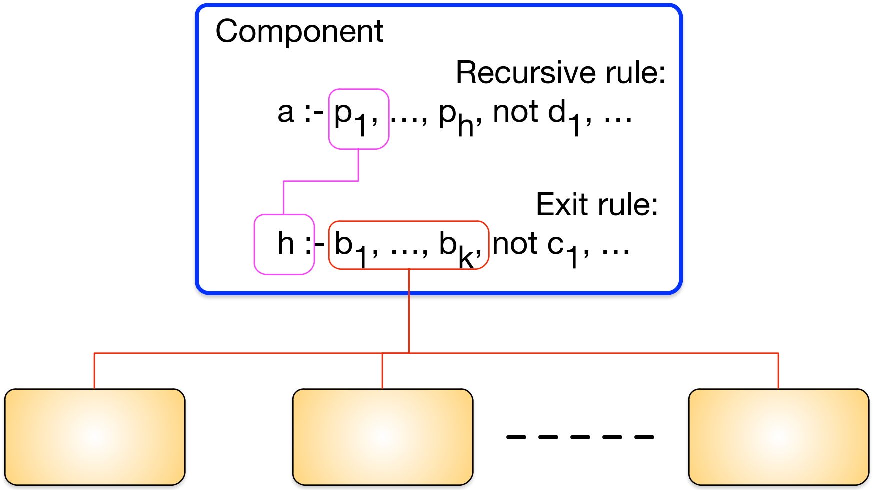
Simona Perri, Francesco Ricca, Marco Sirianni: Parallel instantiation of ASP programs: techniques and experiments. TPLP 13(2): 253-278 (2013)

DLV: Parallel Grounding

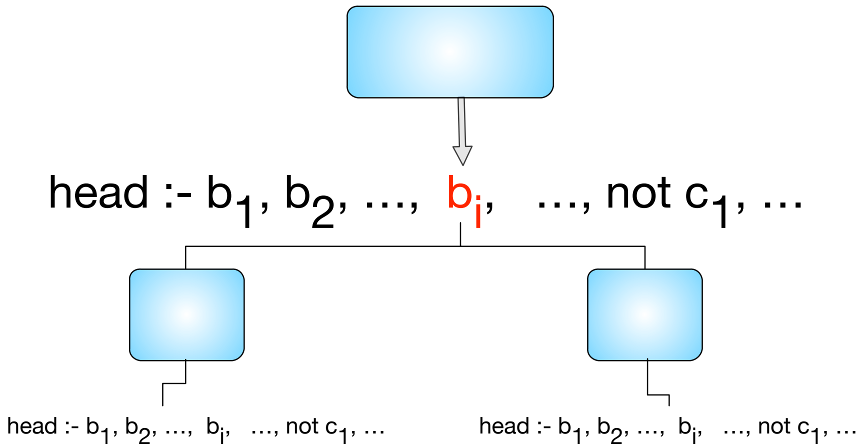
- ASP solvers use ground programs
- Three levels of parallelization of grounding
  1. **Components**:
     - Strongly connected components of the predicate dependency graph (components)
     - Topological sorting guides grounding
     - Components that do not have dependencies can be grounded in parallel

DLV

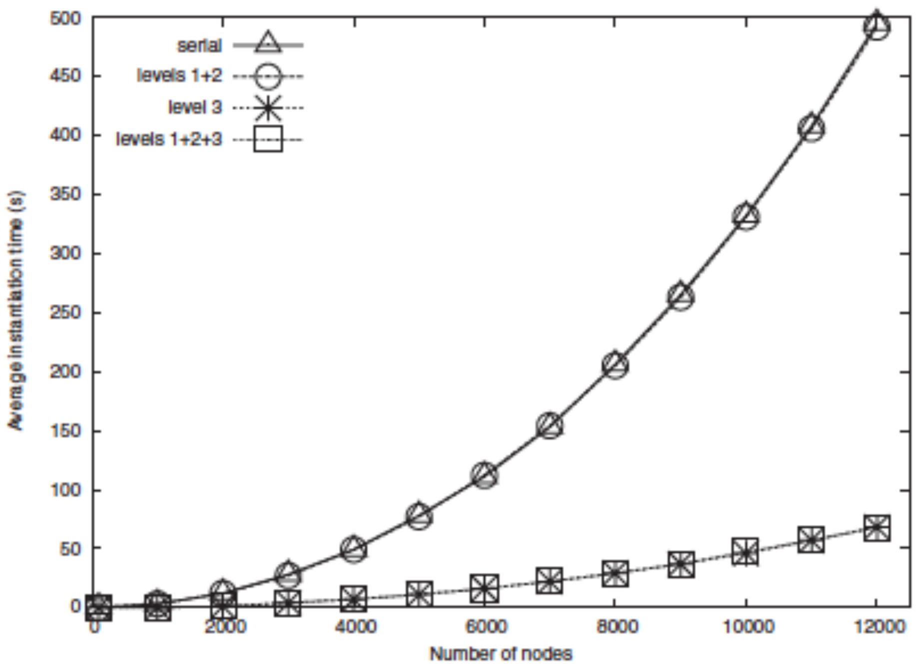
2. Rules within one component



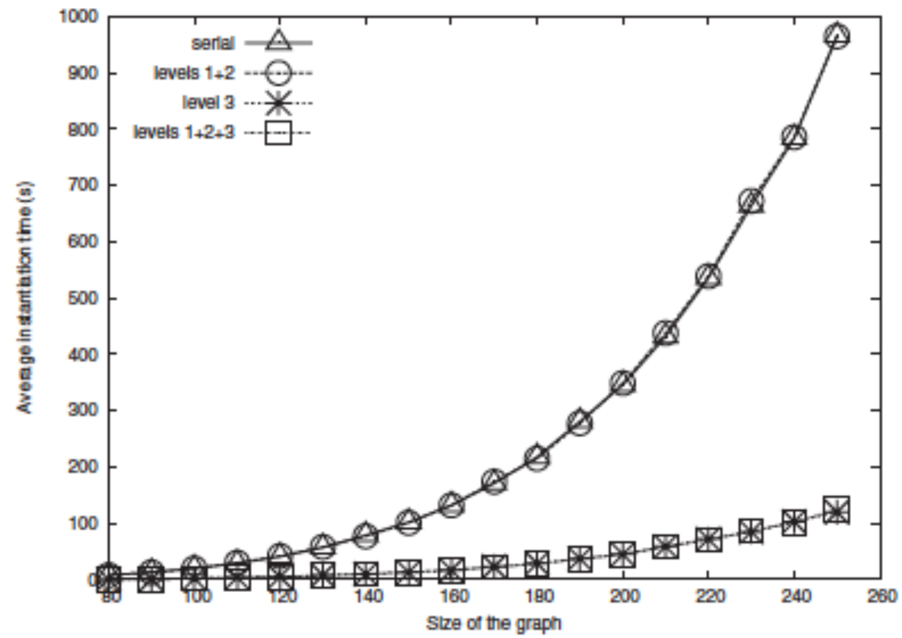
3. Single Rule



DLV



(b) Instantiation times(s) - Hamiltonian Path



(c) Instantiation times(s) - 3-Colorability

Tutorial Roadmap

Basics

Prolog

Systems

Ciao

clasp

YAProlog

ASP

Going Large

Going Small

YESTERDAY

TODAY

(EARLY) TOMORROW

Going BIG

- Issues of size are becoming common
  - ASP with its grounding requirements
    - E.g., encoding Biochemical Pathway planning benchmarks
    - ASP can ground only instances with less than 70 actions (instances 1-4)
    - Out of memory from Instance 5 (163 actions)
  - Use of LP techniques for processing knowledge bases (e.g., RDF stores)
    - E.g., CDAOStore, 957GB, 5 Billion RDF triples
- Distribution as a way to scale
  - Challenging to distribute data and ensure reliability

Big Data

- Distributed File Systems
  - Global file namespace
  - Google GFS, Hadoop HDFS, …
  - Replication for seamless recovery from disk/machine failures
- Chunk Servers
  - Files split into chunks (16-64MB)
  - Chunks replicated (2x or 3x) to different racks
  - Chunk servers are also compute servers

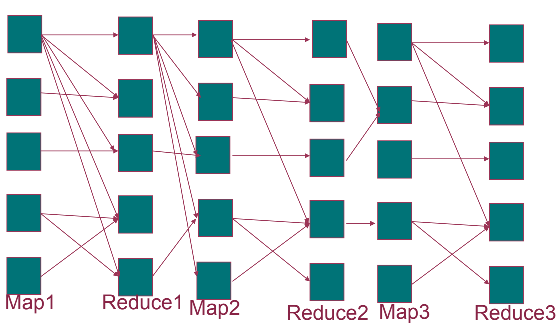
MapReduce and Friends

- Programming models
  - Designed to operate on
- MapReduce

```
Map(Long key, String Value):
        forall word in Value do
                emit(word, 1);
        endforall

Reduce(String key, Iterator Values)
        int count=0;
        forall val in Values do
                count+=val
        endforall
        emit(key, count);
```

Input from DFS



From Natural Joins to Datalog

- $p(X,Y)$:- $q(X,Z)$, $r(Z,Y)$.
  - Map produces:
    - [z, (x,q)] for each incoming q(x,z) fact
    - [z, (y,r)] for each incoming r(z,y) fact
  - Reduce
    - Input: (z, L) where L=[(x,q), (x',q), (y,r), (y',r),…]
    - Output: p(x,y) for each (x,q) ∈ L and (y,r) ∈ L

F. Afrati, J. Ullman. 2010. Optimizing Joins in a Map-Reduce Environment. ACM EDBT, ACM Press.

From Natural Joins to Datalog

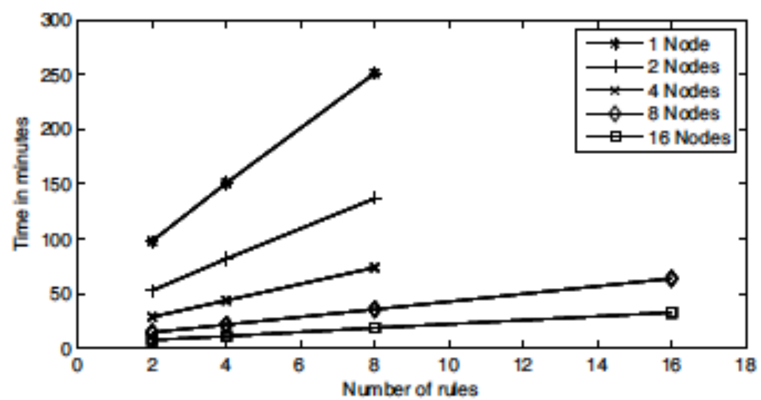
- p(X,Y) :- q(X,Z), r(Z,T), s(T,Y).
  - Assume k = z * t reducers
  - Map: Tuple r(b,c) generates key-value:
    [($hash_z$(b),$hash_t$(c)),r]
  - Map: Tuple q(a,b) generates key-values:
    [($hash_z$(b), k), (a,q)] for each k=1,…,t
  - Map: Tuple s(c,d) generates key-values:
    [(k, $hash_t$(c)), (d,s)] for each k=1,…,z
  - Reducer (i,j): for each (a,q), (d,s), r produce
    p(a,d)

From Natural Joins to Datalog

- Additional considerations
  - HaLoop: iteration of MapReduce reducing communication
  - Set of tasks for each rule in the program
  - Need to add another layer of MapReduce to remove duplicates
    - MAP: for each p(a,b) generate (p,hash(a,b))
    - Reduce: stores (a,b), forwards it only the first time

Specialized applications

- WebPIE
  - RDFS and OWL-Horst Reasoning
    - RDFS: only 2 subgoals in each rule; many are small
    - RDFS: can order rules to reduce number of iterations
- Defeasible Logic
  - Stratified Datalog
  - Ordered rules with defeasible conclusions
  - One set of tasks for each strata
    - First MapReduce task to determine rules that fire
    - Second MapReduce task to apply defeasibility principles



Well-Founded Semantics

- WFS
  - Logic Programming with negation as failure
    p(X) :- a(X), not b(X)
  - Partial Interpretation:
    - Consistent set of literals (e.g., p(a), not b(c), …)
  - Extended Immediate Consequence Operator

$$T_{P,J}(I) = \{A \mid A :- Body \in ground(P), pos(Body) \subseteq I, neg(Body) \cap J = \emptyset\}$$

  - Alternating fixpoint

$$K_0 = lfp(T_{P^+}) \qquad U_0 = lfp(T_{P,K_o})$$

$$K_{i+1} = lfp(T_{P,U_i}) \quad U_{i+1} = lfp(T_{P,K_{i+1}})$$

  - Fixpoint $(K_i,U_i)=(K_{i+1},U_{i+1})$
    - $W^* = K_i \cup \{\text{not } A \mid A \notin U_i\}$

I. Tachmazidis, G. Antoniou, W. Faber "Efficient Computation of the Well-Founded Semantics over Big Data." TPLP 14(4-5): 445-459 (2014)

WFS and MapReduce

- $T_{P,J}(I)$: MapReduce for a typical rule

```
q(X,Y) :- a(X,Z), b(Z,Y), not c(X,Z).
```

- I={a(1,2), a(1,3), b(2,4), b(3,5)}   J={c(1,2)}
- 2-Phase Computation
  1. Positive Part Join – standard 2-way or multi-way join; use tuples from I
     - Map: produces
       <2,(a,1)> <3,(a,1)> <2,(b,4)> <3,(b,5)>
     - Reduce:
       receives <2, [(a,1), (b,4)]> <3, [(a,1), (b,5)]>
       produces ab(1,2,4) and ab(1,3,5)

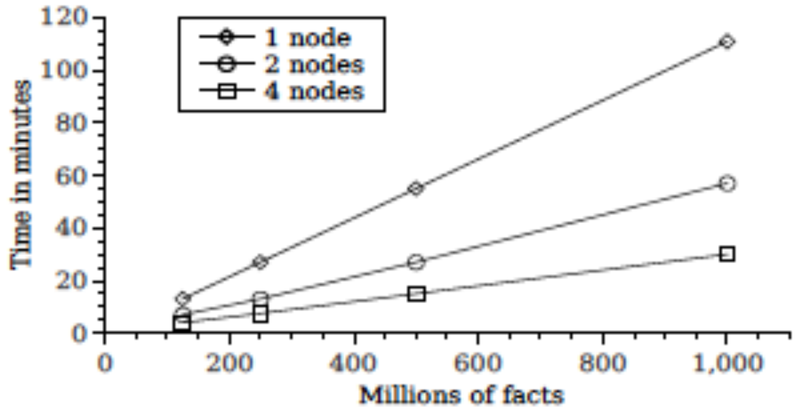
WFS and MapReduce

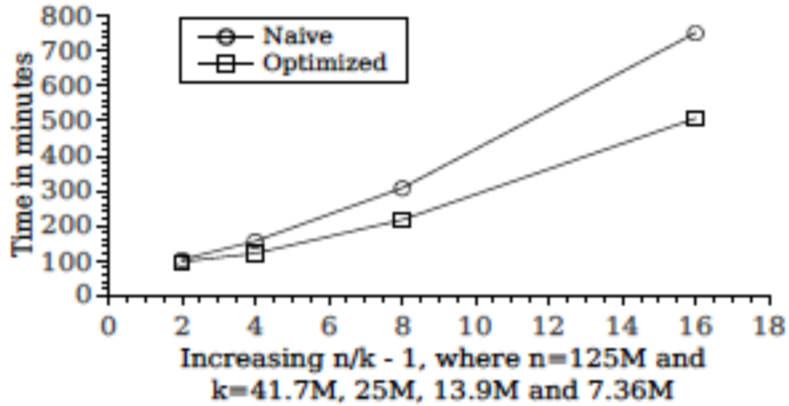
2. Anti-Join: Use tuples from first step and tuples from J

```
q(X,Y) :- ab(X,Z,Y), not c(X,Z).
```

- Map: produces
  <(1,2),(ab,4)> <(1,3),(ab,5)> <(1,2),(c)>
- Reduce:
  receives <(1,2), [(ab,4),(c)]> and <(1,3),[(ab,5)]>
  produces abc(1,3,5)

WFS and MapReduce





win(X) :- move(X,Y), not win(Y)

Cyclic facts:
move(1,2), move(2,3),…move(n,1)

tc(X,Y) :- par(X,Y).
tc(X,Y) :- par(X,Z), tc(Z,Y).
par(X,Y) :- b(X,Y), not q(X,Y).
par(X,Y) :- b(X,Y), b(Y,Z), not q(Y,Z).
q(X,Y) :- b(Z,X), b(X,Y), not q(Z,X).

Chain facts:
b(i,i+k) for $1 \leq i \leq n$

Towards ASP

- Computation view of ASP:
  - Computation: sequence of sets of atoms $X_0=\emptyset \subseteq X_1 \subseteq X_2 \subseteq \ldots$
  - Properties
    - Revision: $X_i \subseteq T_P(X_{i-1})$
    - Convergence: $\bigcup_{i\geq 0} X_i = T_P\left(\bigcup_{i\geq 0} X_i\right)$
    - Persistence: $p \in X_i \setminus X_{i-1}$ then there is a rule p:-Body such that $X_j \models$ Body for each $j \geq i$
  - M is an answer set iff there computation that converges to M

Tutorial Roadmap

Basics

YESTERDAY

Prolog

TODAY

Systems

clasp

YAProlog

ASP

(EARLY) TOMORROW

Going Large

Going Small

GPGPU

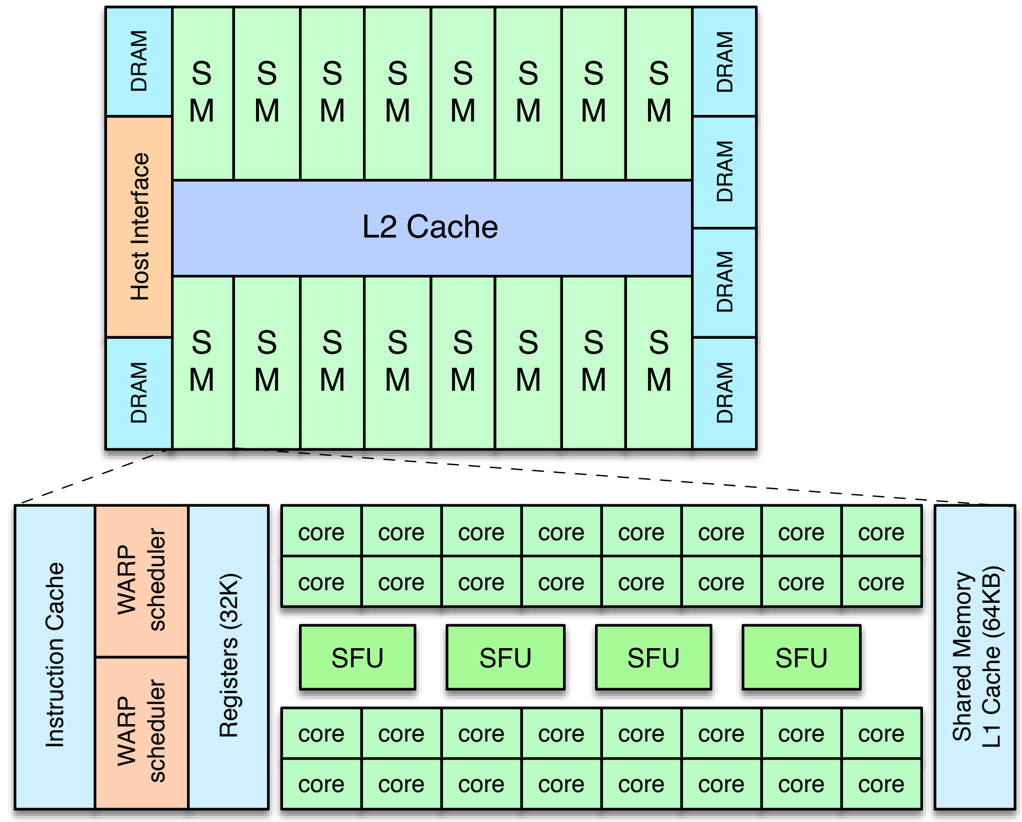
- GPUs
  - Highly parallel architectures
  - Inexpensive
- GPGPU: General Purpose GPU
  - Vendors provide APIs and programming frameworks for general purpose applications
  - Use GPUs as massively parallel architectures for general purpose computing
  - OpenCL
  - Compute Unified Device Architecture (CUDA)

CUDA

- Designed for data oriented applications
- Heterogeneous serial-parallel computing
- C for CUDA – extension to C
- SIMT – Single Instruction Multiple Thread
  - Same instruction executed by different threads
  - Data might be different from thread to thread

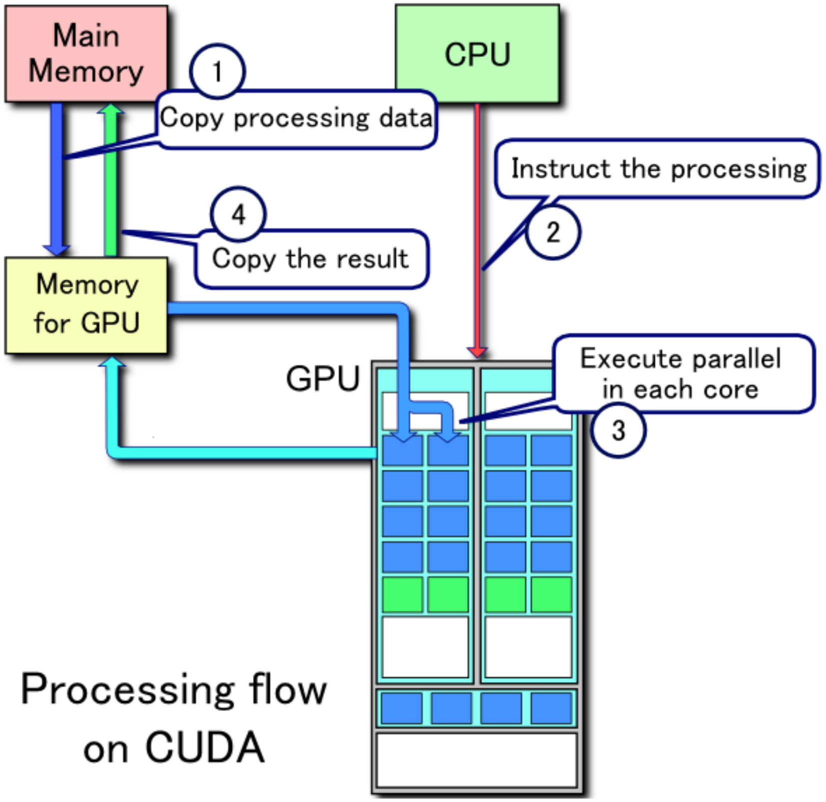
CUDA

- The "physical" view



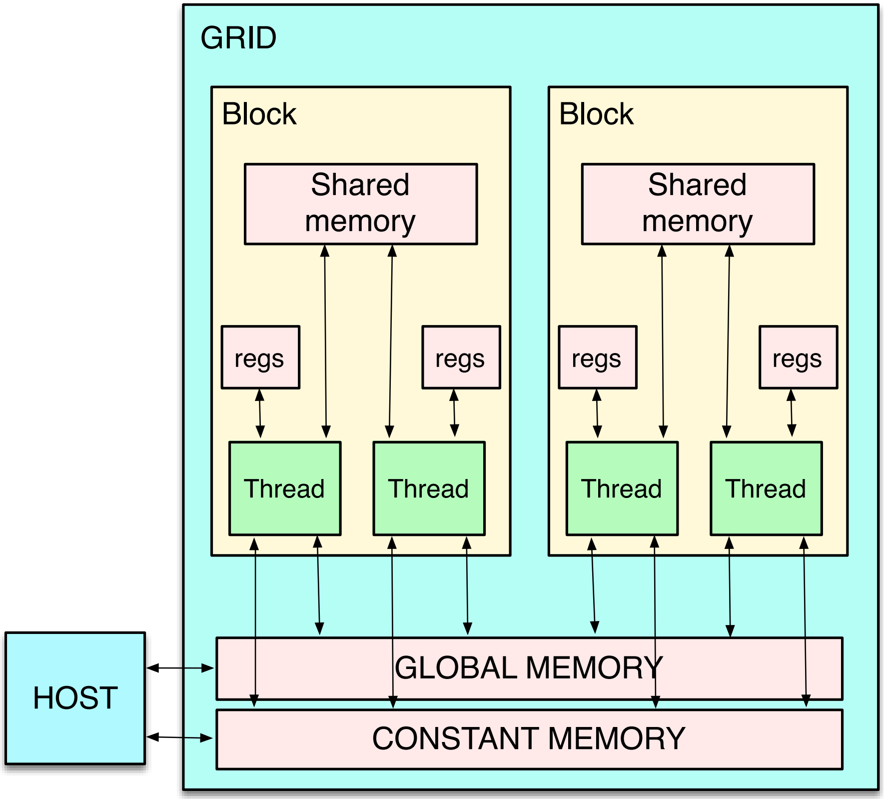
CUDA

- The “logical” view
  - Hybrid program
    - Host functions: executed on the CPU (__host__)
    - Kernels: executed on the GPU (__global__ or __device__)
  - Programmer responsible for
    - Determine threads to be launched on the GPU
    - Data organization
      - E.g., __device__ or __shared__
    - data movements between CPU and GPU
      - cudaMemcpy
    - Synchronization, memory management, …
      - synchtreads()



CUDA

- Kernel executed by many threads
  - Very lightweight
  - Fast context switch
- Threads organization
  - 2D collection of threads (Block)
    - Threads can synchronize
    - Threads can use shared memory
  - 3D collection of blocks (Grid)
    - Blocks can interact through Global memory



GPU and LP

- Very limited applications of GPU-level parallelism directly to LP
  - But growing fast…

Datalog in CUDA

- Datalog execution as relational algebra operations

selection

`example(Z,X) :- table1(Y,X), table2(Y,Z,a), table3(Z,B,C).`

Join over Y

`example(Z,X) :- temptable1(Y,X,Z), table3(Z,B,C).`

Join over Z

`example(Z,X) :- temptable2(Y,X,Z,B,C).`

Projection over Z,X

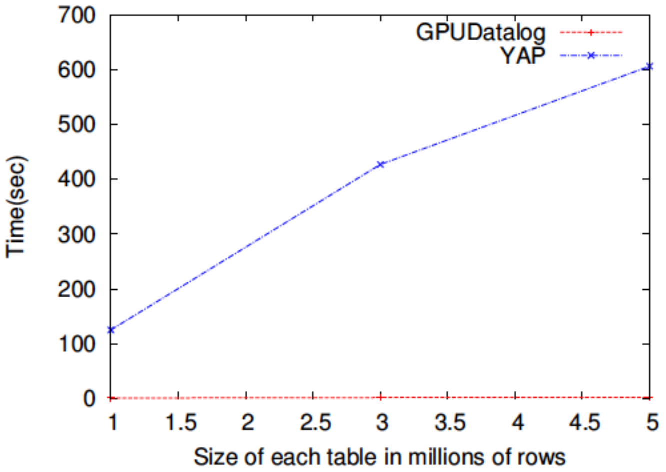
`example(Z,X).`

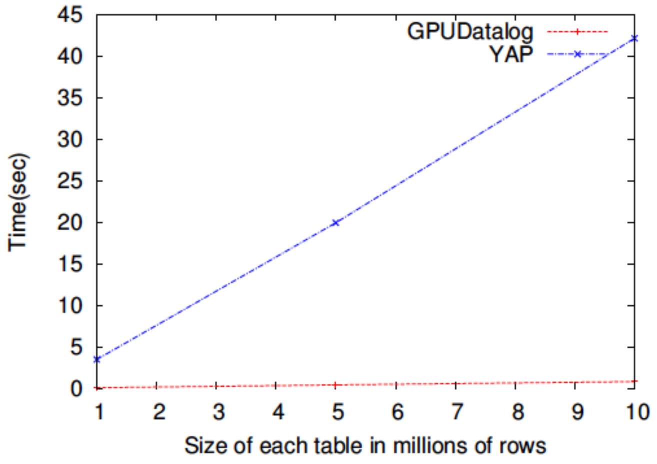
C. Martinez-Angeles, I. Dutra, V. Santos Costa, J.Buenabad-Chavez, 2013. A Datalog Engine for GPUs. KDPD: 152-168

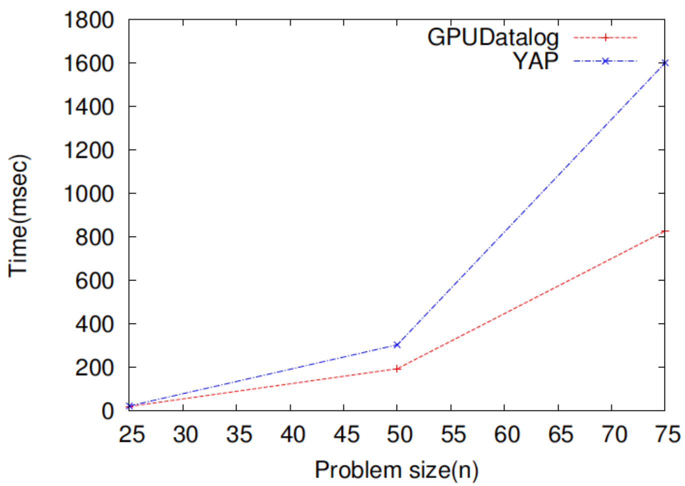
Datalog in CUDA

- Host:
  - Maintain facts in global memory
  - Explicit memory management
    - List with least recently used facts at the end
- *Selection*: 3 kernels
  1. Mark all rows that satisfy selection condition
  2. Count marked rows (using prefix sum)
  3. Write results in global memory (use results of prefix sum as indices)
- *Projection*:
  - 1 kernel, copy rows
- *Join*:
  - Extract arrays of two columns to join
  - Sort one and create a CSS-Tree for it
  - Search tree to determine join positions
  - First join will count successful joined elements
  - Second join will write the results

Datalog in CUDA







More General LP

- Not there yet
- But…
  - Many components have been investigated
  - Applied to similar frameworks
  - Work is in progress…

GPGPU and Search

- Parallelizing depth-first search (e.g., Prolog)
  - Distributing the actual search is challenging
    - Lots of subtrees, high memory cost, no coalescing
  - Need many threads for hiding memory latencies

	Backtracking	GPU
Problem Instance	Irregular Access	Regular access, locality
Work Unit	Memory, Computation Variable	Constant size, perfect SIMD
Output	Exponential Size (enumerate); hard to estimate	Polynomial size, a-priori
Search Space	Tree-based, unbalanced	Fixed, a-priori

J. Jenkins, I. Arkatkar, J.D. Owens, A.N. Choudhary, N.F. Samatova: Lessons Learned from Exploring the Backtracking Paradigm on the GPU. Euro-Par (2) 2011: 425-437

GPGPU and Search

- Most successful search problems on GPUs
  - Ability to remove stack and perform breadth-first traversal
  - Ability to exploit fine-grained parallelism within each node
    - Maintain a depth-first exploration, e.g.,
      - Construction of next states (parallel maximal cliques enumeration)
      - Evaluate bounds (B&B)

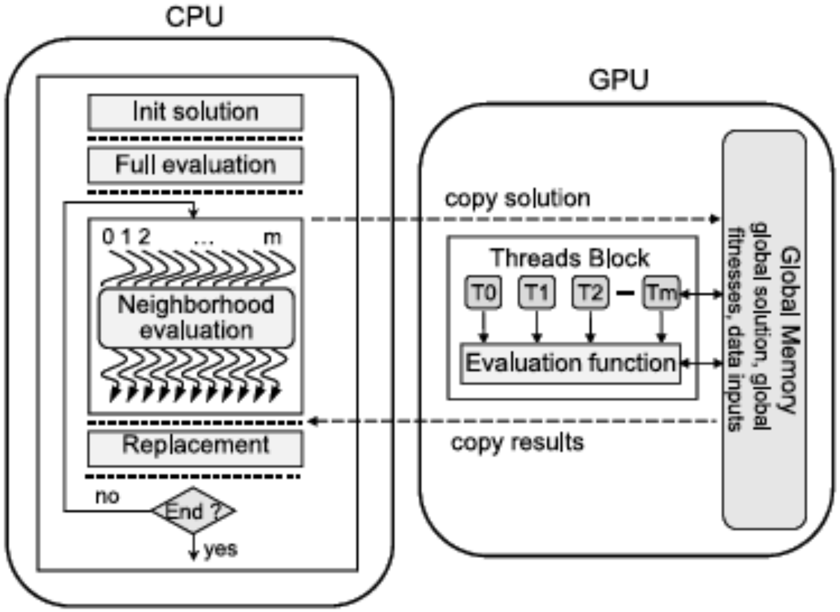
Problem Size	CPU time (s)	GPU time (s)	Speedup
100	1.59	0.41	3.84
200	4.85	0.91	5.33
300	9.82	1.44	6.80
400	10.94	1.27	8.61
500	13.39	1.44	9.27

	Ava80	Slp	rmat1	rmat2
CPU 1-core	3.6	15.7	24.6	108
CPU 4-core no lb	1.2	5.1	13.8	59
CPU 4-core lb	1.1	3.8	8.19	33.2
GPU	0.9	11.2	10.8	60.5

GPGPU and Search

- Effective for enhanced local search

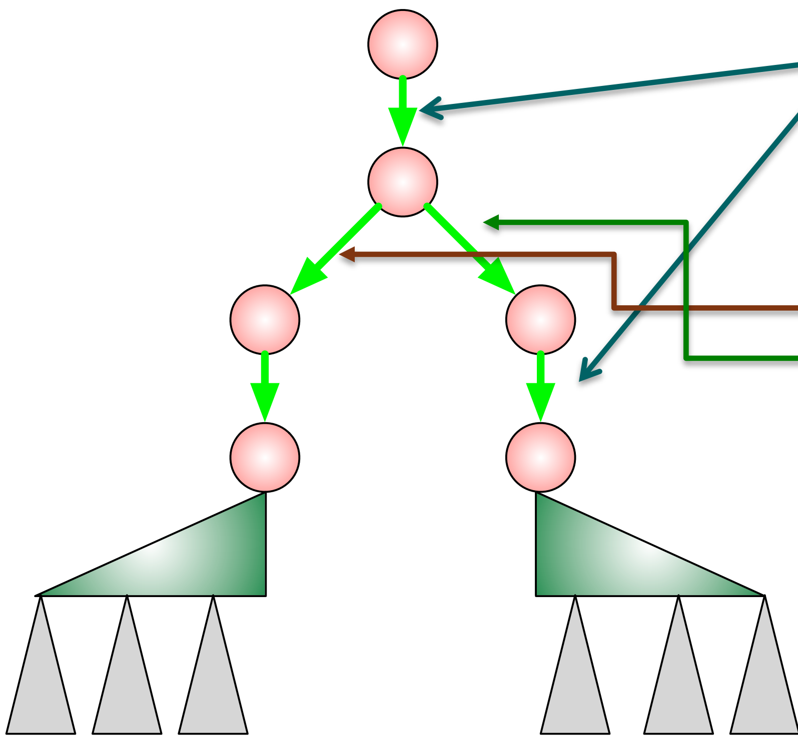
	Hamming Distance 1			Hamming Distance 2		
Instance	CPU	GPU	Speedup	CPU	GPU	Speedup
121	1.4	1.5	0.9	106	5.2	20.4
151	2.1	1.7	1.2	193	8.0	24.1
171	2.7	1.9	1.4	305	11.3	26.9
201	3.8	2.2	1.7	455	17.6	29.5



N. Melab et al. "ParadisEO-MO-GPU: a Framework for Parallel GPU-based Local Search Metaheuristics" GECCO, ACM Press, 2013.
T. Van Luong et al. "A GPU-based Iterated Tabu Search for Solving the Quadratic 3-dimensional Assignment Problem", AICCSA, IEEE Press, 2010.

GPGPU and SAT

- Parallelizing DPLL



DPLL(ϕ,ϑ):

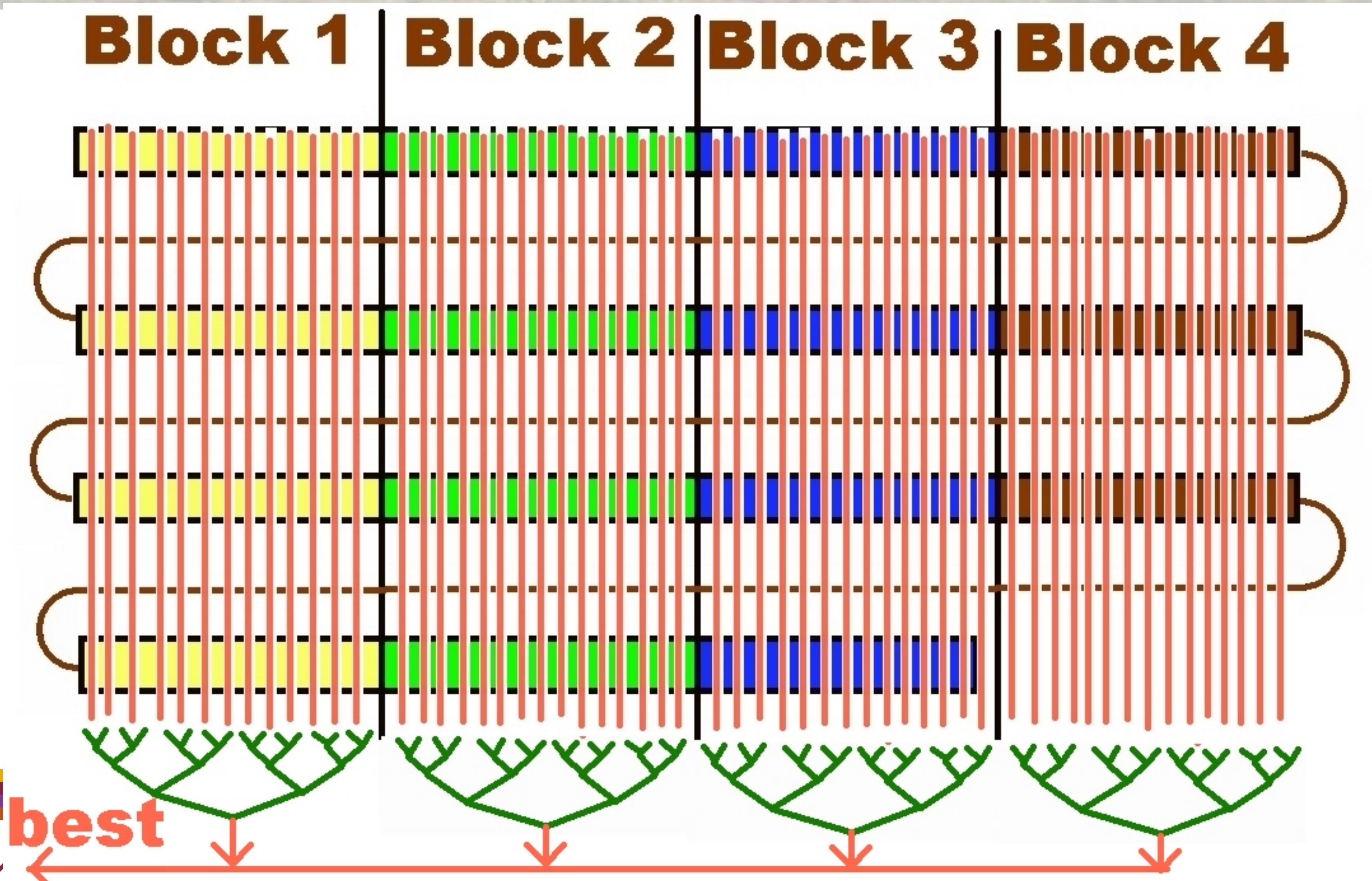
ϑ' = unit_propagation(ϕ,ϑ)
if (satisfied($\phi\vartheta$')) then
  return ϑ'
else if (falsified($\phi\vartheta$')) then
  return FALSE
else
  X = select_variable(ϕ,ϑ'))
  ϑ_1 = ϑ U {X/TRUE};
  ϑ_2 = ϑ U {X/FALSE};
  if (DPLL(ϕ,ϑ_1)) then
    return ϑ_1
  else return DPLL(ϕ,ϑ_2))

A. Dal Palu, A. Dovier, A. Formisano, E. Pontelli. (to appear).
CUD@SAT: SAT Solving on GPUs.
Journal of Experimental and Theoretical AI.

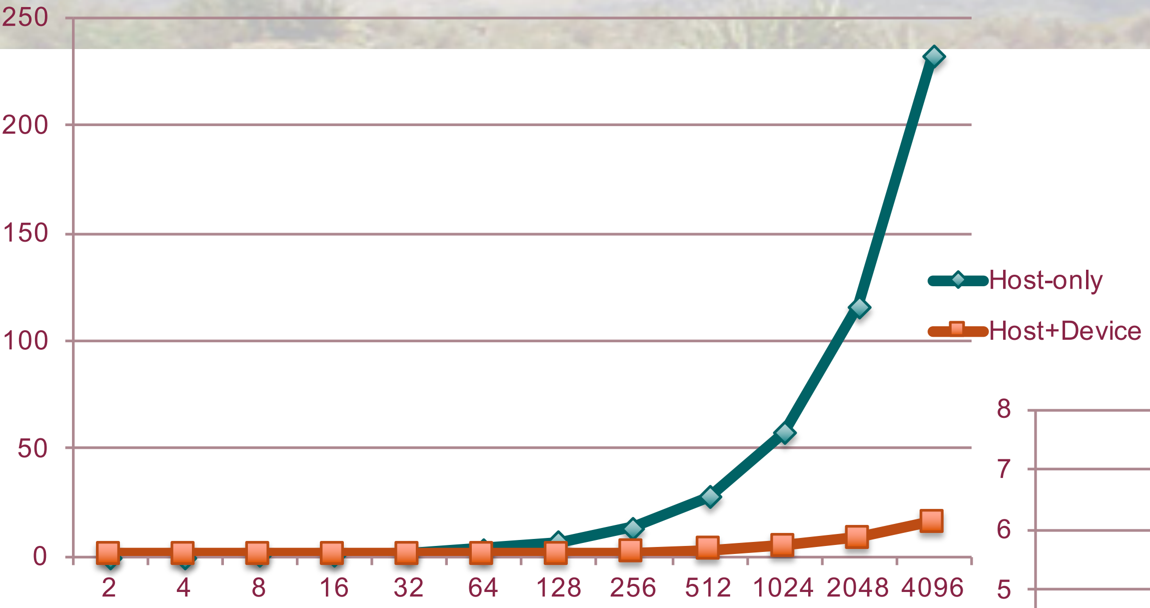
GPGPU and SAT

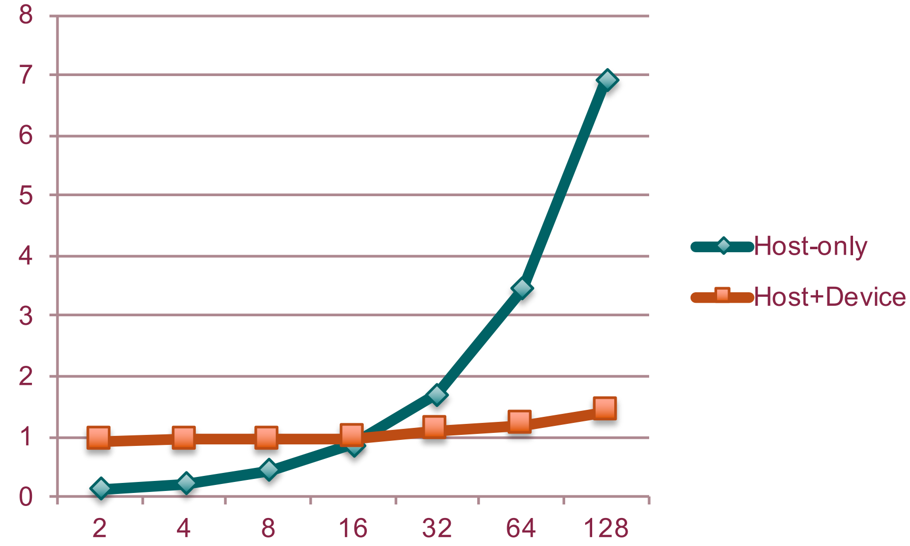
- Majority of the efforts
  - E.g., MESP (miniSAT Enhanced with Survey Propagation)
- Parallelizing Unit Propagation
  - Given a partial assignment ϑ: mask array
    - mask[i]=0 if clause i is satisfied by ϑ
    - mask[i]=-1 if clause i is falsified by ϑ
    - mask[i]=u if there are u > 0 unknown literals in clause i and ϑ does not satisfy the clause
  - mask_prop procedure: returns
    - -1 if there is a value of i such that mask[i]=-1
    - 0 if mask[i]=0 for all clauses
    - Pointer to an unknown literal in clause i where mask[i]>0 and mask[i] is minimal among all those with mask[i]>0

GPGPU and SAT



GPGPU and SAT



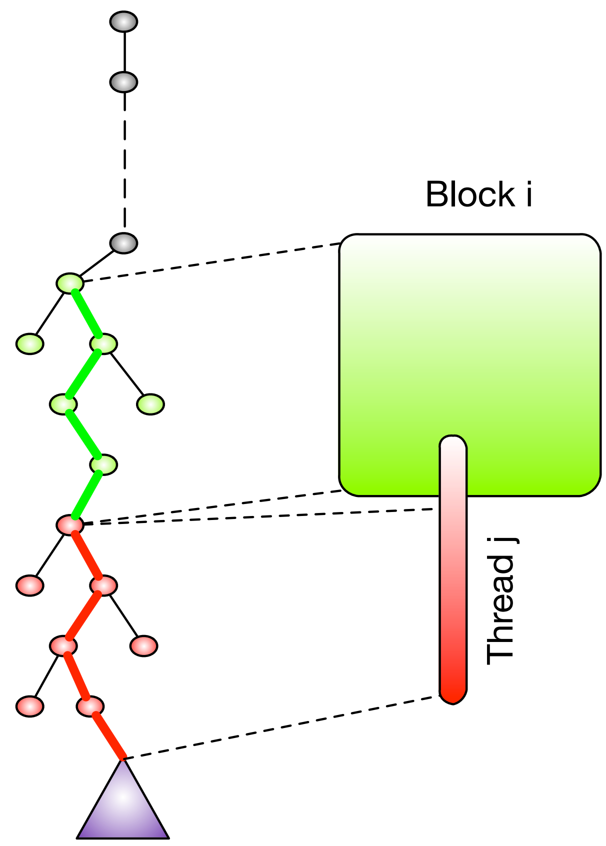


GPGPU and SAT

- Parallelizing Search
  - Focus on tail of the search
  - If/when the formula (reduced by current ϑ) is *"large but not huge"*, we can parallelize the search in it

GPGPU and SAT

- Idea:
  - **MaxV** variables undefined (sorted)
  - First *$log_2(B)$* variables are deterministically assigned in each block
    - All threads in one block assign same truth value to such variables
  - Next *$log_2(T)$* variables are deterministically assigned in each thread
  - Thread performs an iterative DPLL on the remaining *$MaxV-log_2(B)-log_2(T)$* variables



GPGPU and SAT

Benchmark	Host-only	Vars	Clauses	Speed-up	MaxV-B-T
marg3x3add8.shuffled-as.sat03-1449	1242	41	224	88.3	35-6-7
marg3x3add8ch.shuffled-as.sat03-1448	1751	41	272	92.7	35-6-7
battleship-5-8-unsat	2.26	40	105	7.1	40-8-8
battleship-6-9-unsat	69.38	54	171	6.2	54-10-8
unif-k5-r21.3-v50-c1065-S1449708927-022	220.92	50	1065	12.9	50-6-7
unif-k5-r21.3-v50-c1065-S370067727-038	213.95	50	1065	11.4	50-7-7
sgen1-unsat-61-100	440.49	61	132	7.8	61-7-7
Jnh16	53.14	100	850	4	100-8-8

Towards GPGPU and ASP

- Compute assignments
  $A \subseteq \{Tp \mid p \text{ is an atom}\} \cup \{Fp \mid p \text{ is an atom}\}$
- Computation based on
  - Selection
  - Propagation
    - Based on nogoods Δ_{Π} for a program Π
      (set of literals that cannot be extended into an answer set)
    - Two classes of nogoods
      - Completion nogoods
      - Loop nogoods
    - A violates nogood δ if $\delta \subseteq A$
    - A is an answer set of program Π iff A is a solution of Δ_{Π}

F. Vella, A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli: CUD@ASP: Experimenting with GPGPUs in ASP solving. CILC 2013: 163-177

Towards GPGPU and ASP

- From a sequential ASP solver to a GPU-solver

```
A = ∅; dl = 0; Δ_Π=Parse(Π);
loop
    conflict = NoGoodsCheck(A, Δ_Π)
    if (conflict ∧ (dl=0)) then return No Answer Set
    if (conflict ∧ dl > 0) then
        (dl,δ) = ConflictAnalysis(A, Δ_Π)
        Δ_Π = Δ_Π ∪ δ
        A = A ∖ { p ∈ A| dl < dl(p) }
    else if there is δ∈Δ_Π such that δ∖A={p} then
        A = UnitPropagation(A, Δ_Π)
        Δ_Π = UnfoundedSetCheck(A, Δ_Π)
    else if unassigned atoms > 0 then
        A = Select(A)
    else return A
endloop
```

```
A = ∅; dl = 0; Δ_Π=Parse(Π);
loop
    conflict = NoGoodsCheck(A, Δ_Π)
    if (conflict ∧ (dl=0)) then return No Answer Set
    if (conflict ∧ dl > 0) then
        (dl,δ) = ConflictAnalysis(A, Δ_Π)
        Δ_Π = Δ_Π ∪ δ
        A = A ∖ { p ∈ A| dl < dl(p) }
    else if there is δ∈Δ_Π such that δ∖A={p} then
        A = UnitPropagation(A, Δ_Π)
        Δ_Π = UnfoundedSetCheck(A, Δ_Π)
    else if unassigned atoms ≥ k then
        A = Select(A)
    else if 0 < unassigned atoms < k then
        A = ExhaustiveSearch(A)
        if StableTest(A, Π) then return A
        else Δ_Π = LearnNoGoods(A, Π)
    else return A
endloop
```

Towards GPGPU and ASP

Problem	GT250	GT460	C2075	K20c	K80	Titan	Titan X
0001-visitall-14-1	128	93	70	46	34	14	13
0007-graph colouring-125-0	214	155	134	91	64	66	29
0023-labyrinth-11-0	TO	899	TO	314	51	51	49
0167-sokoban-15-1	102	40	33	59	63	71	28

Problem	Smodels	Cmodels	Clasp-None	Clasp	Yasmin
channelRoute_3	2.08	1.42	69.27	0.24	0.37
Knights_17	0.91	1.99	0.05	0.06	0.16
Knights_20	9.61	3.85	0.22	0.2	0.46
Schur_4_42	0.07	0.6	0.02	0.05	0.07

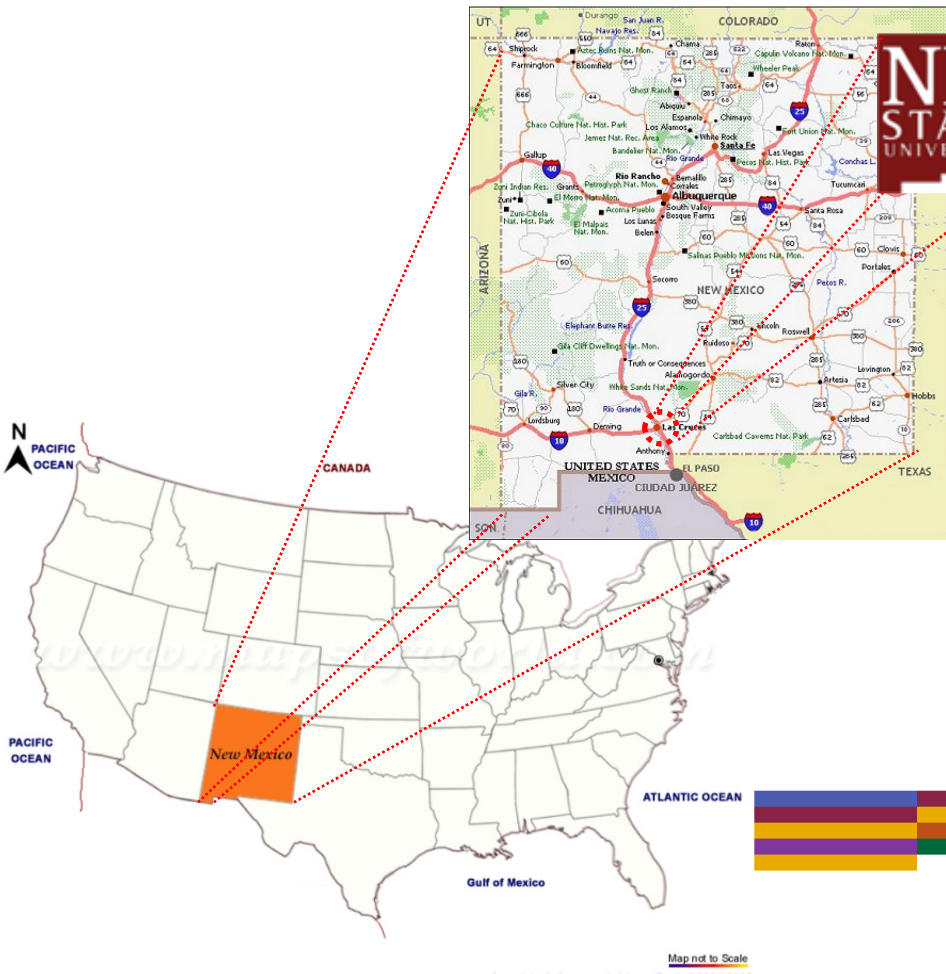
CONCLUDING REMARKS

In summary…

- Decades of research on extracting parallelism from logic based paradigms
- Research has informed developments in many related areas
- Many opportunities
  - Novel applications with high performance demands
  - Avoids many challenges present in other paradigms
  - Features suitable to parallelization, e.g.,
    - Search and non-determinism
    - Language features (e.g., map, list processing)
- Many challenges
  - Memory management
  - Granularity
  - Static analysis

Acknowledgments

- KLAP = Knowledge representation, Logic, and Advanced Programming



G. Gupta
U.T.Dallas

M. Hermenegildo
U.Politecnica Madrid

H. Viet Le
Iowa State U.

A. Dovier
Univ. Udine

A. Formisano
Univ. Perugia

A. Dal Palu
Univ. Parma

T. Son
NMSU

Thank You

Questions?

Logic Programming

- Definite programs (Pure Prolog, Datalog)
  - Collection of first-order Horn clauses

```
reachable(X) :- edge(Y,X), reachable(Y).
```

$$\forall X, Y(edge(X,Y) \wedge reachable(Y) \rightarrow reachable(X))$$

  - Declarative semantics based on least Herbrand model

Logic Programming: Prolog

- Typical Operational Semantics: SLD Resolution
  - Top-down, goal oriented
- Language enriched with extra-logical constructs
  - I/O and other side effects
  - Control operators (e.g., cut, oneof, freeze)
  - Embedding of other "pre-interpreted" constructs

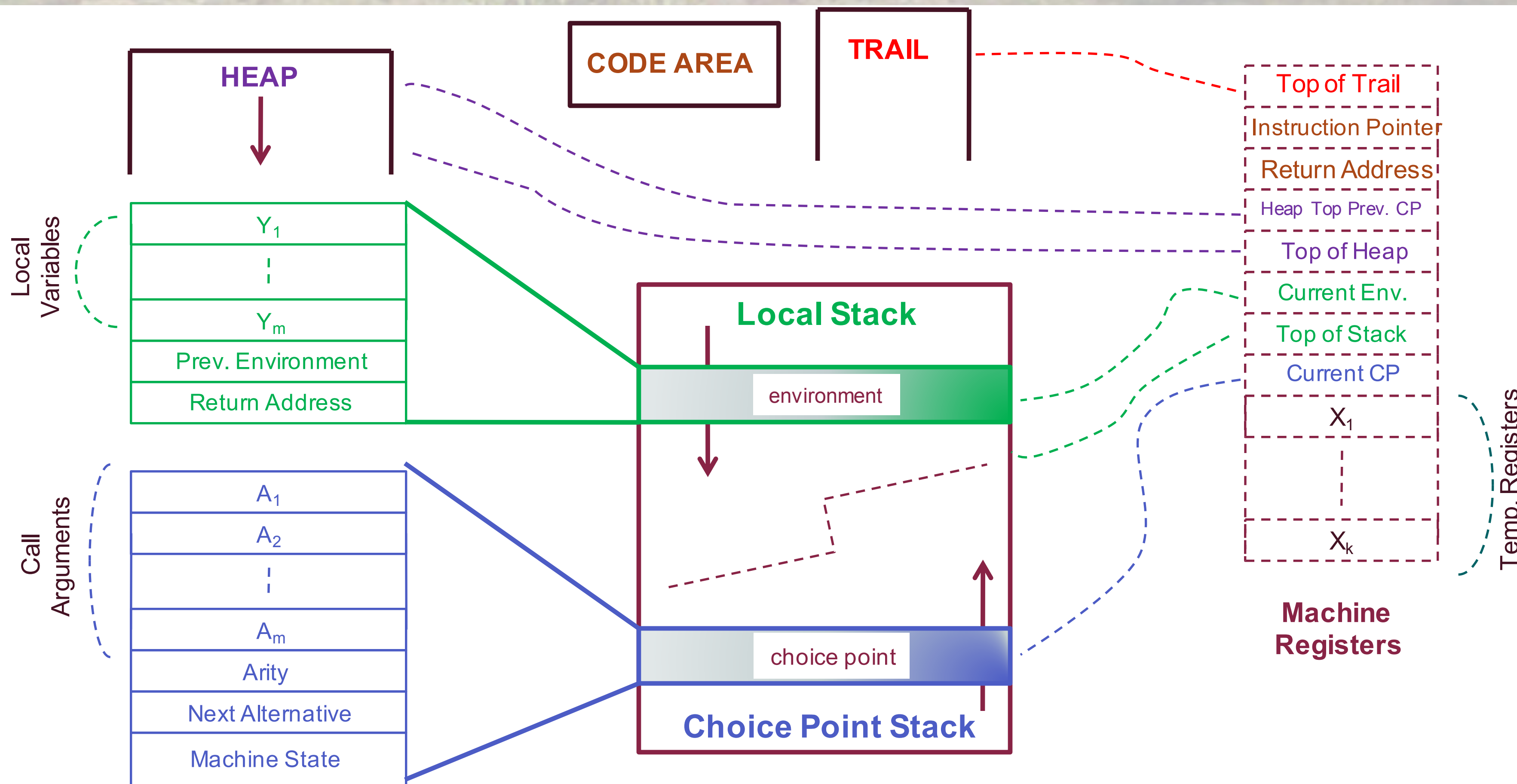
```
a(X) :- write(X), nl, c(X).
```

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
```

```
a(X,Y) :- X:1..4, X+Y#>0.
```

- Compiled-based implementations
  - Warren Abstract Machine (WAM)

Logic Programming: WAM



Logic Programming

- Normal programs
  - enter negation as failure

```
color(X,red) :- node(X), not color(X,blue).
```

  - Alternative semantics
    - Well-founded semantics [XSB, tabling]
    - Answer set semantics [Answer Set Programming]
  - Answer Set Programming
    - Program = modeling of problem
    - Solutions = answer sets of the program
    - Execution Models
      - bottom-up execution models (each solution = one answer set)
        - » variations of DPLL
        - » mapping to SAT