Parallel Answer Set Programming

Agostino Dovier[1] Andrea Formisano[2] Enrico Pontelli[3]

1. Università di Udine

2. Università di Perugia

3. New Mexico State University

PCR'17 @ CADE'17 — Gothenburg, August 2017

Material from *Handbook of Parallel Constraint Reasoning*, ch.7.
Youssef Hamadi and Lakhdar Sais (eds.), Springer, 2017

Answer set programming

- A successful form of logic programming paradigm
- Knowledge representation and Non-monotonic reasoning (Horn + default negation)
  - Logical theories serve as problem specifications
  - Solutions are described by models of the theories
- Strong theoretical foundation: it originates from extensive research on semantics of LP with negation
- Expressive power: it captures (in its simplest form) the *NP* complexity class
- Efficient inference engines

ASP Programs

An ASP program Π is a collection of propositional rules of the form

$$r: \quad p_0 \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n$$

- p_0 and $\{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ are denoted by $\mathit{head}(r)$ and $\mathit{body}(r)$, resp.
- $\{p_1, \ldots, p_m\}$ is denoted by $\mathit{body}^+(r)$
- $\{p_{m+1}, \ldots, p_n\}$ is denoted by $\mathit{body}^-(r)$

The positive dependence graph $\mathcal{D}^+_\Pi = (V, E)$ of Π is such that

- The set of nodes is $V = \mathit{atom}(\Pi)$
- The set of edges is $E = \{(\mathit{head}(r), q) \mid r \in \Pi, q \in \mathit{body}^+(r)\}$

Π is tight if $\mathcal{D}^+_\Pi$ contains no trivial cycles

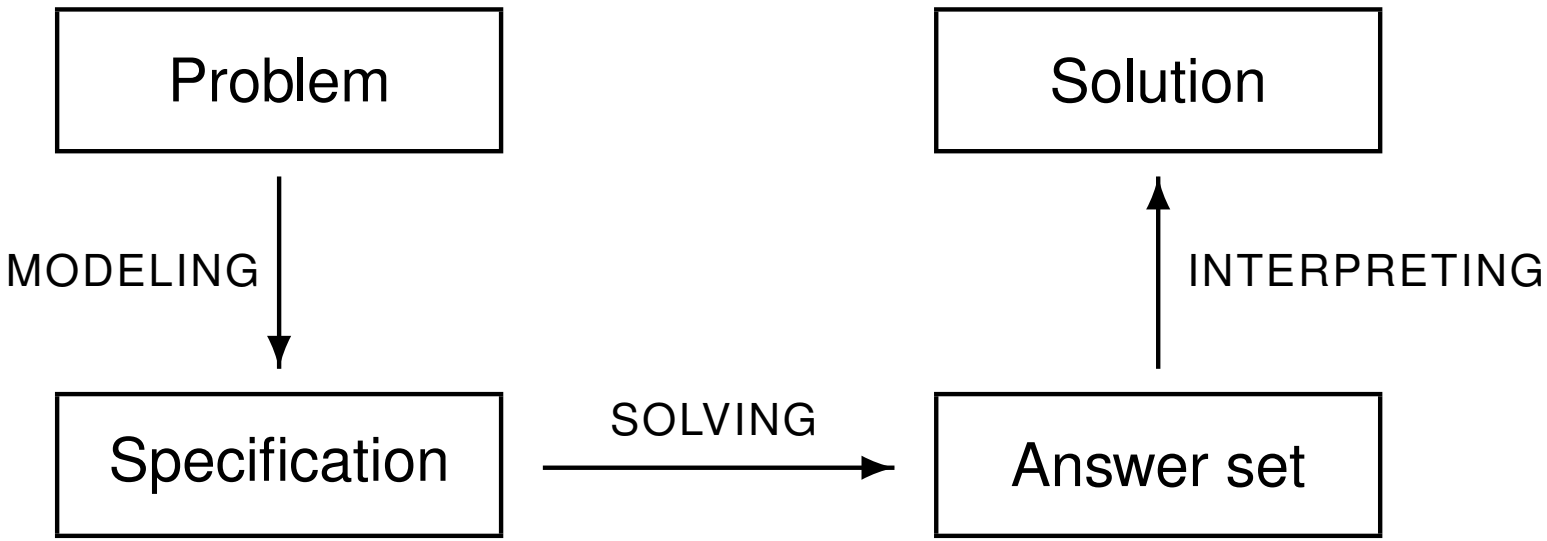
ASP Programs

- Semantics of ASP program Π is given in terms of answer sets (or *stable models*)
- A set M of atoms is an answer set for Π if it is the least Herbrand model of the reduct Π^M obtained by
  - removing from Π all rules r such that $M \cap body^-(r) \neq \emptyset$; and
  - removing all negated atoms from the remaining rules

Model-based problem solving in ASP

Typical approach in ASP:

- An ASP program (i.e., a logical theory) serves as problem specification
- Each answer set encodes a solution



Example: Hamiltonian cycles

```
% Graph:
node(0).    node(1).    node(2).    node(3).  ...
edge(0,1).  edge(0,2).  edge(1,3).  edge(1,2).  ...

% Choice: select those edges that are "in" the solution
 in(A,B) :- node(A), node(B), edge(A,B), not out(A,B).
out(A,B) :- node(A), node(B), edge(A,B), not in(A,B).

% Each node is traversed once:
false :- node(A), node(B), node(C), B!=C, in(A,B), in(A,C).
false :- node(A), node(B), node(C), A!=B, in(A,C), in(B,C).

% Each node is reachable (from 0) using the selected edges:
reach(A) :- node(A), in(0,A).
reach(A) :- node(A), node(B), reach(A), in(A,B).
false :- node(A), A!=0, not reach(A).
```

The atoms of the form `in(n,m)` in an answer set describe a solution of the problem

Note: variables range over the set of constants of the program

Computation of answer sets

Problem: The definition of answer set is non-constructive

It involves a guess&check procedure:

- guess a candidate set of atoms M
- compute the least model of Π^M
- check if such model is M.

Real ASP-solvers exploit more effective computational models

- `smodels`: `select_atom()` + `expand()`
- `cmodels`: completion + SAT-solving
- `clasp`: nogood-driven DPLL-like (with unfounded-set check)
- `yasmin`: nogood-driven DPLL-like (ASP-computation)
- MIP, SMT, ...

Smodels approach

`Smodels` computes the answer sets of a program Π by alternating

- non-deterministic choices of literals to be set true (i.e. to be included in a partial interpretation of Π): `select_atom()`
- deterministic expansion (enforcing stability) of the current partial interpretation: `expand()`

Algorithm 2: Basic SMODELS Procedure

```
1  procedure compute(P: Program; S: Interpretation)
2  S' ← expand(P,S);
3  if ¬consistent(S') then
4      return False;
5  if complete(S') then
6      return S';
7  ℓ ← select_atom(P,S');
8  S' ← compute(P,S' ∪ {ℓ});
9  if S' = False then
10     return compute(P,S' ∪ {¬ℓ});
11 else
12     return S';
```

ASP-solving via SAT-solving (cmodels)

- Given a program Π consider its completion Π_{cc}:

$$\Pi_{cc} = \left\{\beta_r \leftrightarrow \bigwedge_{a \in body^+(r)} a \wedge \bigwedge_{b \in body^-(r)} \neg b \mid r \in \Pi\right\} \cup \left\{p \leftrightarrow \bigvee_{r \in body_\Pi(p)} \beta_r \mid p \in atom(\Pi)\right\}$$

- the answer sets of Π are minimal models of Π_{cc}
- loop formulas must be considered to rule-out unsupported models of Π_{cc}
- `cmodels` exploits a SAT-solver to determine minimal models of Π_{cc} and *lazily* generates loop formulas

Nogoods and conflict-driven solvers (clasp)

- A nogood is a forbidden set/conjunction of literals
- Π_{cc} can be "compiled" into a collection $\Delta_{\Pi_{cc}}$ of completion nogoods of the forms:
  - $\{not\ \beta_r\} \cup \{a \mid a \in body^+(r)\} \cup \{not\ b \mid b \in body^-(r)\}$
  - $\{\beta_r, not\ a\}$ for each $a \in body^+(r)$ and $\{\beta_r, b\}$ for each $b \in body^-(r)$

  for each r in Π, and
  - $\{not\ p, \beta_r\}$ for each $r \in body_\Pi(p)$, for each head p in Π
  - $\{p\} \cup \{not\ \beta_r \mid r \in body_\Pi(p)\}$, for each head p in Π
- similarly one introduces loop nogoods to reflect loop-formulas

The state-of-the-art ASP-solver `clasp` uses a conflict-driven DPLL-like procedure and fruitfully adapts SAT-technology (conflict analysis, nogood learning, backjumping, forgetting, ...)

Loop nogoods

Given a program Π, an assignment A, and a set of atoms U

- the set of external bodies for U is defined as
$$EB_\Pi(U) = \{\beta_r \mid r \in \Pi, body^+(r) \cap U = \emptyset\}$$
- U is unfounded w.r.t. A, if, for each rule $r \in \Pi$, it holds that:
$head(r) \notin U$, or $body(r)$ is falsified by A, or $body^+(r) \cap U \neq \emptyset$
- loop nogoods correspond to loop formulas. For each $p \in U$, we have the nogood:
$$\{p\} \cup \{not\ \beta_r \mid \beta_r \in EB_\Pi(U)\}$$
- The set of loop nogoods is denoted by Λ_Π.
Let $\Delta_\Pi = \Delta_{\Pi_{cc}} \cup \Lambda_\Pi$.

An alternative ASP-computation (yasmin)

An ASP-computation is a sequence of sets of atoms $I_0 = \emptyset, I_1, I_2, \ldots$ s.t.

- $I_i \subseteq I_{i+1}$ for all $i \geq 0$ (Persistence of Beliefs)
- $I_\infty = \bigcup_{i=0}^{\infty} I_i$ is such that $T_\Pi(I_\infty) = I_\infty$ (Convergence)
- $I_{i+1} \subseteq T_\Pi(I_i)$ for all $i \geq 0$ (Revision)
- if $p \in I_{i+1} \setminus I_i$ then there is a rule $p \leftarrow body$ in Π such that $I_j \models body$ for each $j \geq i$ (Persistence of Reason)

(where T_Π is the usual *immediate consequence operator* of definite LP)

Prop: M is an answer set of Π iff there exists an ASP-computation that converges to M, namely, $M = \bigcup_{i=0}^{\infty} I_i$

The `yasmin` prototype adopts a nogood-based approach to develop an ASP-computation, avoiding the introduction of loop-nogoods

Variables in rules?

- Semantics of ASP is defined for propositional programs, but
- variables may occur in ASP rules:

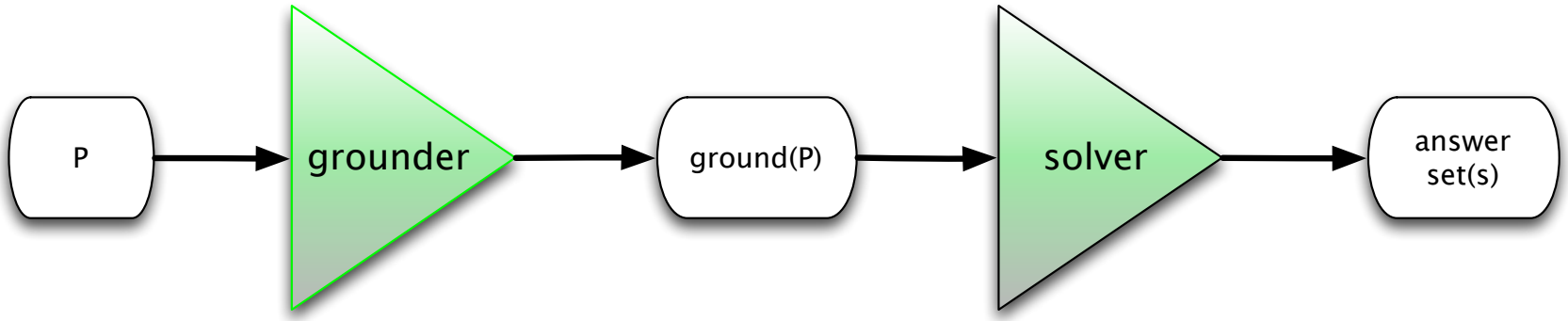
```
 ...
 in(A,B) :- node(A), node(B), edge(A,B), not out(A,B).
out(A,B) :- node(A), node(B), edge(A,B), not in(A,B).
 ...
```

- They act as placeholders to be replaced, in each possible way, by any constant defined in the program
- (Almost) all ASP-solver deal with propositional programs only
- The answer sets of a non-ground program are defined as the answer sets of its grounding
- Hence, a grounding step has to be performed before solving

ASP solving pipeline

Solvers are usually paired with grounders: `lparse+smodels`; `gringo+clasp`; `dlv` and its (integrated) grounder; ...

The classical ASP solving pipeline:



In what follows we will review some of the techniques introduced in the literature

- to parallelize the grounding step
- to parallelize the solving step

Parallel Grounding

Parallelizing `lparse`

A first attempt in parallelizing the `lparse` grounder is described in [BPEL05]:

- A distributed implementation on a Beowulf cluster
- N “agents” organized as a master-slave structure
- Master agent partitions the program rules and assigns each part to a slave agent
- Load balancing through estimations of the expected number of ground rules
- Each slave agent computes its portion of the grounded program
- The master collects and merges the results

Parallel grounding in DLV (1)

The multi-level parallel grounder of DLV [CPR08]. Key ideas:

- By exploiting the dependency graph of the program (and its SCCs), split the program in components
- Perform grounding of the components in topological order (first level of parallelism)
- Within each component, threads are spawn to process rules in parallel (second level of parallelism)
- For each rule one or more threads might be spawn to perform a portion of its grounding (third level of parallelism)
- The tasks are split/distributed among threads by considering estimations of the sizes of the expected results, size of variables' domains, selectivity of variables, hardness of rules, ...
- Load balancing and granularity control are adjusted dynamically (useful for recursive rules)

Parallel grounding in DLV (2)

The multi-level parallel grounder of DLV [CPR08].

- The target architecture is a multicore/multiprocessor system (SMP architecture)
- Threads (actually, Posix threads) cooperate through shared memory
- Synchronization achieved through barriers (thread_join)
- Instead of creating/terminating each thread, a thread pool is managed

[CPR08] reports on extensive experimentation

- higher performance achieved w.r.t. the serial counterpart
- great scalability of the approach on varied collection of problem instances

Parallel ASP-Solving

Parallelizing `smodels`-like solvers

Let us recall the basic algorithm implemented in `smodels`:

Algorithm 2: Basic SMODELS Procedure

```
1  procedure compute(P: Program; S: Interpretation)
2  S' ← expand(P,S);
3  if ¬consistent(S') then
4      return False;
5  if complete(S') then
6      return S';
7  ℓ ← select_atom(P,S');
8  S' ← compute(P,S' ∪ {ℓ});
9  if S' = False then
10     return compute(P,S' ∪ {¬ℓ});
11 else
12     return S';
```

The search proceeds by constructing/exploring a binary search tree (branching correspond to calls to `select_atom()`)

Parallelizing `smodels`-like solvers

In the `smodels` procedure there are two sources of nondeterminism

- `select_atom()`:
  different selections correspond to different paths in the search space (*don't know*)
- `expand()`:
  various rules to be used in completing the expansion (*don't care*)

Parallelism can be exploited in both cases

Let us focus on the former

Parallelizing `smodels`-like solvers

[BPEL05] introduces a parallelization of the `smodels` solver

- The target architecture is a shared-memory platform, using Posix threads
- similar proposals (e.g. [FMMT01]) consider message-passing communication models

Let us assume we have

- N "processors" and that (unexplored) sub-tree are initially assigned to each of them
- each processor executes a standard (serial) computation
- processors communicates to coordinate and share work
- a processor might ask for work or allow other processors to explore part of its sub-tree

The overall structure is as follows:

Parallelizing smodels-like solvers

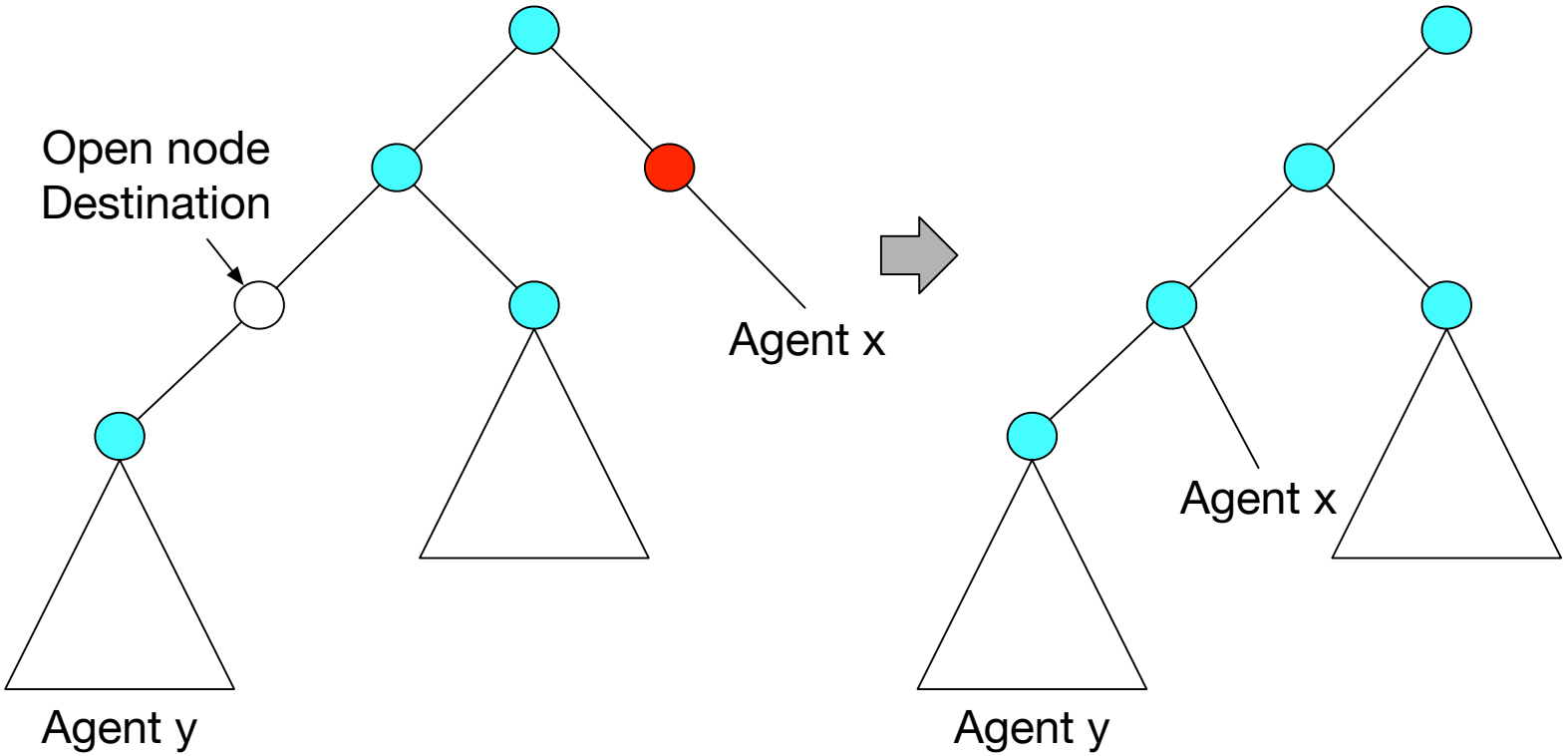
```
1  procedure COMPUTE (P: Program)
2  S ← expand(P, ∅)
3  (Branch, ℓ) ← Select_Private_Node(P, S) /* Initial partition      */
4  while (¬Termination_Detection()) do
5      while (¬Completed_Local_Task(Branch)) do
6          if (Need_to_Schedule()) then /* respond to requests      */
7              Branch ← Scheduling(Branch)
8          Branch ← Branch + [ℓ]; S ← expand(P, S ∪ {ℓ})
9          if (¬consistent(S)) then /* backtrack                    */
10             (ℓ, S, Branch) ← backtrack(P, S, Branch)
11         else if (complete(S)) then
12             Output (S) ;
13             if (¬Complete_Local_Task(Branch)) then
14                 (ℓ, S, Branch) ← backtrack(P, S, Branch)
15         else
16             atom ← select_atom(P, S);
17             CHOICE: ℓ ← atom OR ℓ ← ¬atom /* choice point    */
18         Branch ← Branch + [ℓ]
19     (ℓ, S, Branch) ← Look_for_Work() /* seek work from others    */
```

Parallelizing `smodels`-like solvers

In parallelizing the exploration of the search tree, one has to address two challenges:

- Task sharing:
  how to move a processor to a different part of the search space (once it has completed its sub-task)
- Scheduling:
  how to locate which part of the tree a processor should explore next

Intuition about Task sharing



Task sharing requires the data structures owned by the processor X (the receiver) to be modified to reflect the structure owned by another processor Y (the sender)

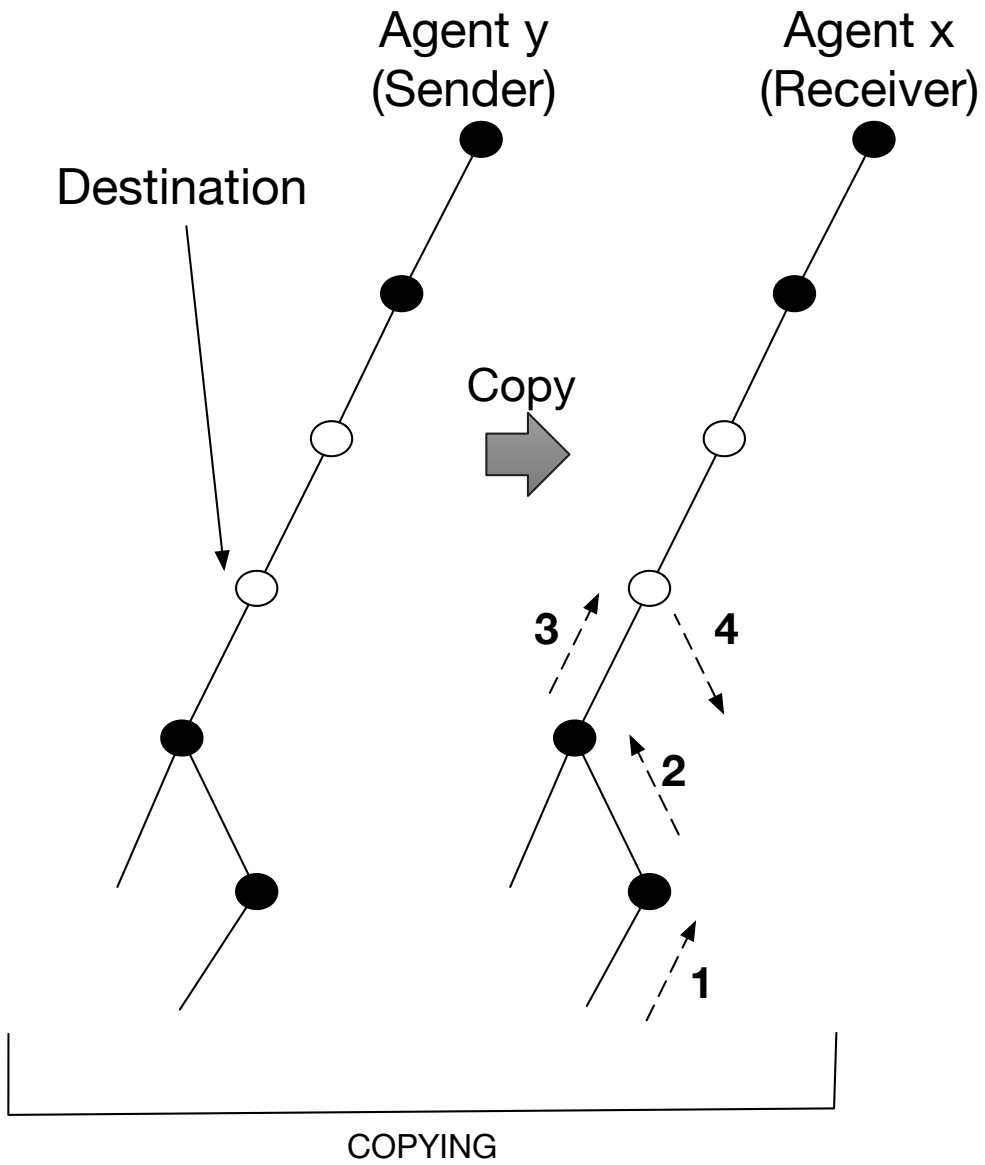
Task-sharing

The most successful options for task sharing proposed in the literature are

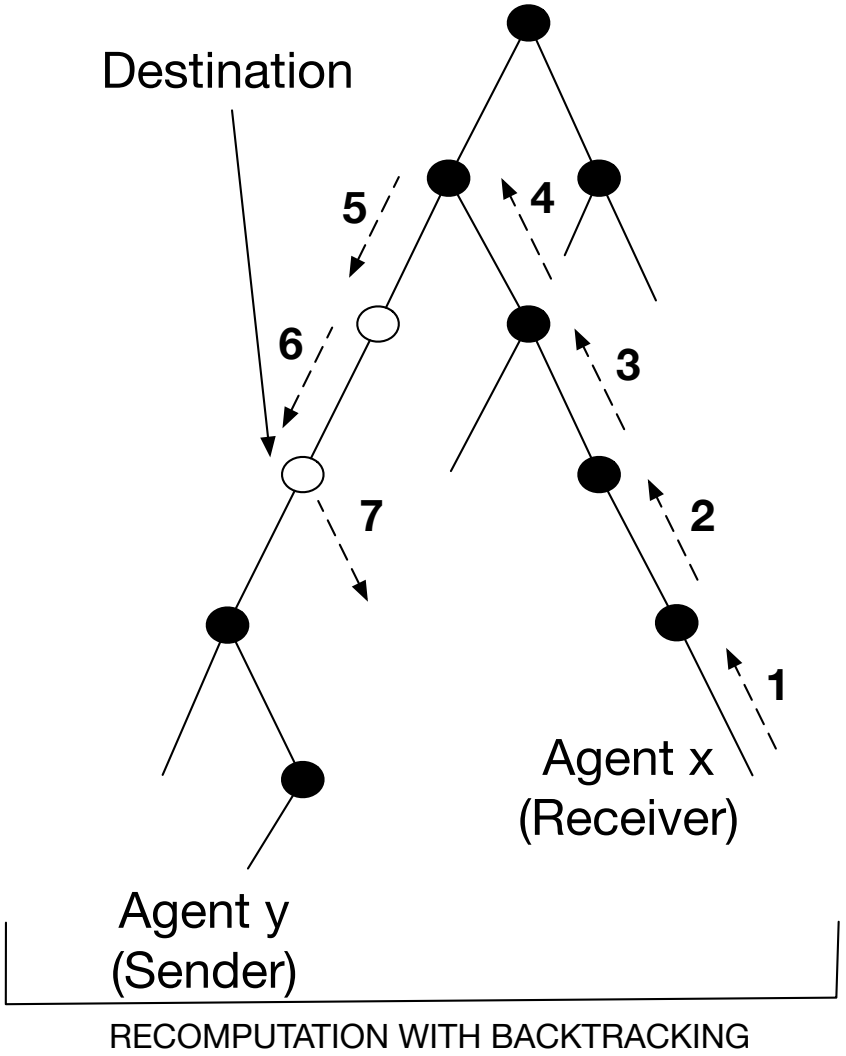
- Coping
- Recomputation with backtracking
- Recomputation without backtracking

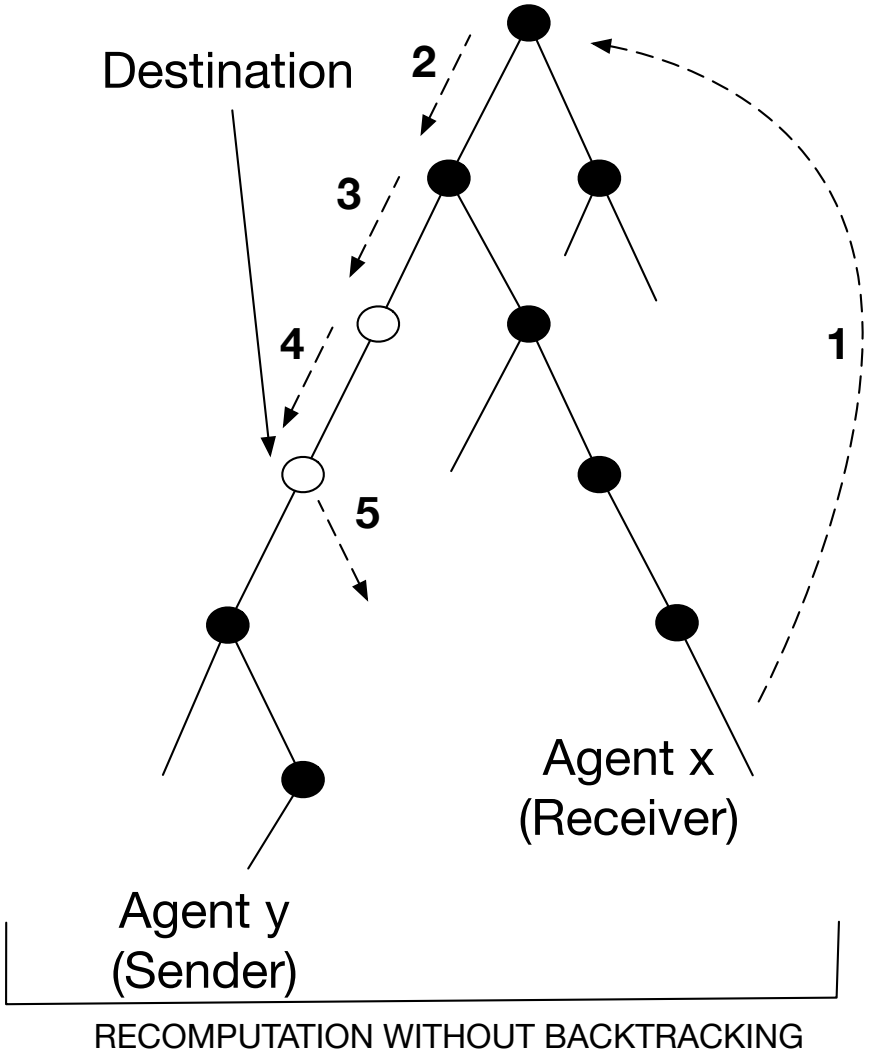
Some more hints...

Task-sharing: Coping



Task-sharing: Recomputation





Task-sharing in practice

- No clear winner, among these options
- Performance depends on
  - the communication model
    (shared-memory `vs` message-passing)
  - the amount of data to be copied/sent
    (the sub-tree `vs` the choice points)
  - the amount of work needed
    (backtrack `vs` recompute)
  - ...

Scheduling

Intuitively, the questions to be answered are:

- who is the sender?
- who is the receiver?
- what is the destination point?
- when a sharing operation has to be performed?

Different possibilities concern:

- Scheduling symmetry: centralized (master-slave) or symmetric
- Scheduling initiation: receiver-initiated (the receiver asks for work) or sender-initiated (the sender ask for helpers)
- Location policy: for example, the destination point is randomly selected or the “closest” point is selected (w.r.t. the amount of work needed to copy, recompute, backtrack)

Scheduling in practice

Considering the experimental results reported on in the literature, one observes that

- the performance of the various options are usually affected by the choice of the benchmarks, the communication model, the underlying architecture, ...
- but, in general, the experimental results seem to indicate a dominance of symmetric scheduling over asymmetric approaches

Parallel lookahead

Another interesting source of parallelism in `smodels` originates from the lookahead strategy:

- before choosing an atom to be decided (`select_atom()`)
- add an undecided literal L to the partial model and perform an `expand()` operation
- if this step originates inconsistency, then deterministically extend the model by adding the complement of L
- repeat the process for each undecided literal

Since all these extensions are deterministic and essentially independent, it is easy to design a parallel version of the lookahead strategy where undecided atoms are assigned to different processors

GPU-based ASP-computation exploiting nogoods

[DFPV16]: the first attempt in exploiting a GPU-based parallelism for ASP-solving

The main goal is to design a solver that:

- exploits GPUs and the CUDA framework
  $\Rightarrow$ massive parallelism for all crucial tasks,...
- adopts a "nogood-driven" approach
  $\Rightarrow$ SAT/ASP technology, heuristics, learning,...
- relies on ASP-computations
  $\Rightarrow$ focus on completion nogoods

Ingredients for a nogood-driven solver

Considering the basic DPLL-like approach exploited in `clasp`, one identifies these crucial tasks:

- Preprocessing: parse the input; compute completion nogoods, dependency graph, statistics for heuristics; data transfer to the device...
- Selection: perform a step in an ASP-computation, to select next branching atom (decision step)
- Propagation: propagate the consequences of decision steps (specific kernels for short nogoods, atom-activity, ...)
- Nogood-Check: look for violations of nogoods
- Conflict-Analysis: in case of conflict, learn new nogoods
- Backjumping: in case a conflicting partial assignment is reached, update device data structures consequently

All Blue tasks can run on the device. The host performs I/O, some preprocessing, data transfers to/from the device

Basic schema of the CUDA application

1: $current\_dl := 1$; $A := \emptyset$ ▷ Initial decision level and assignment
2: $(A, \texttt{Violation}) :=$ **InitialPropagation**(A, Δ)
3: **if** (Violation is true) **then return** no answer set
4: **else**
5: **loop**
6: $(\Delta_A, \texttt{Violation}) :=$ **NoGoodCheckAndPropagate**(A, Δ) ▷ Conflict(s) detection
7: $A := A \cup \Delta_A$;
8: **if** (Violation is true) $\wedge$ $(current\_dl = 1)$ **then return** no answer set
9: **else if** (Violation is true) **then**
10: $(current\_dl, \delta) =$ **ConflictAnalysis**(Δ, A) ▷ Learning (possibly multiple) and
11: $\Delta := \Delta \cup \{\delta\}$; $A := A \setminus \{\overline{p} \in A \mid current\_dl < dl(\overline{p})\}$ ▷ backjump
12: **end if**
13: **if** (A is not total) **then**
14: $(\overline{p}, OneSel) :=$ **Selection**(Δ, A) ▷ Step in ASP-computation
15: **if** ($OneSel$ is true) **then** $current\_dl{+}{+}$; $dl(\overline{p}) := current\_dl$; $A := A \cup \{\overline{p}\}$
16: **else** $A := A \cup \{Fp : p \text{ is unassigned}\}$
17: **end if**
18: **else return** $A^T \cap atom(\Pi)$
19: **end if**
20: **end loop**
21: **end if**

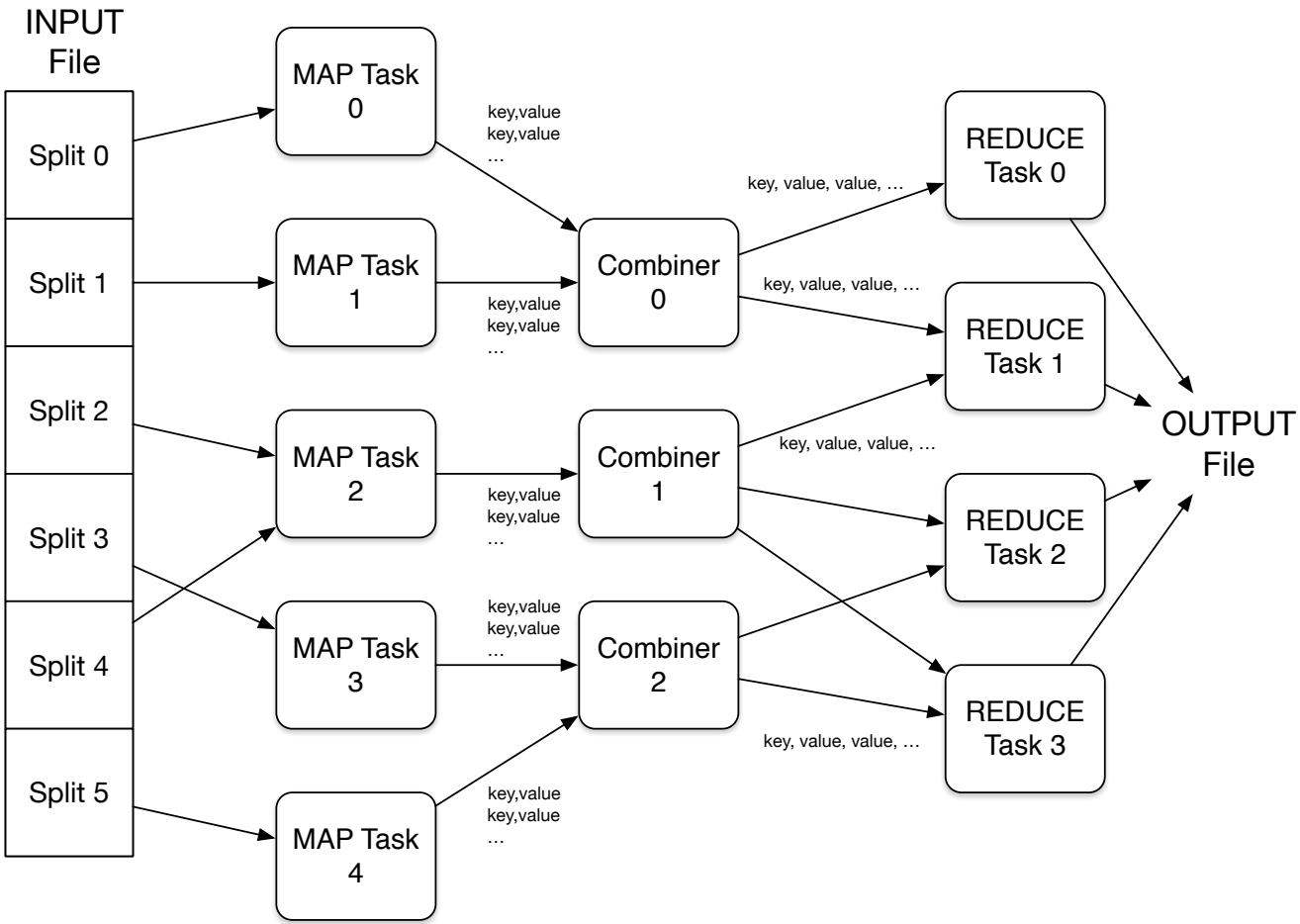
Preliminary promising results

The results of experimentation with different GPUs are encouraging

- performance scales with the computing power of the GPUs
- the current prototype cannot compete with the state-of-the-art solvers
- but much has to be done in improving various aspects of the solver. E.g.:
  - introduce smart heuristics (branching, ...)
  - exploit topological structure of the instance (SCCs, tightness, ...)
  - run multiple concurrent kernels and split search space
  - incorporate ideas used by smodels (seen before), such as lookahead, task sharing, ...
  - move to multi-GPUs, heterogeneous architectures,...
  - ...

Going further

The Map-Reduce programming model



Intuitively: various, distributed, Map and Reduce tasks handle data organized in $\langle key, value \rangle$ pairs. Map tasks produce sets of pairs while Reduce tasks collect values corresponding to keys

Map-Reduce for ASP?

Initial proposals exploiting the Map-Reduce programming model in computational logics appeared

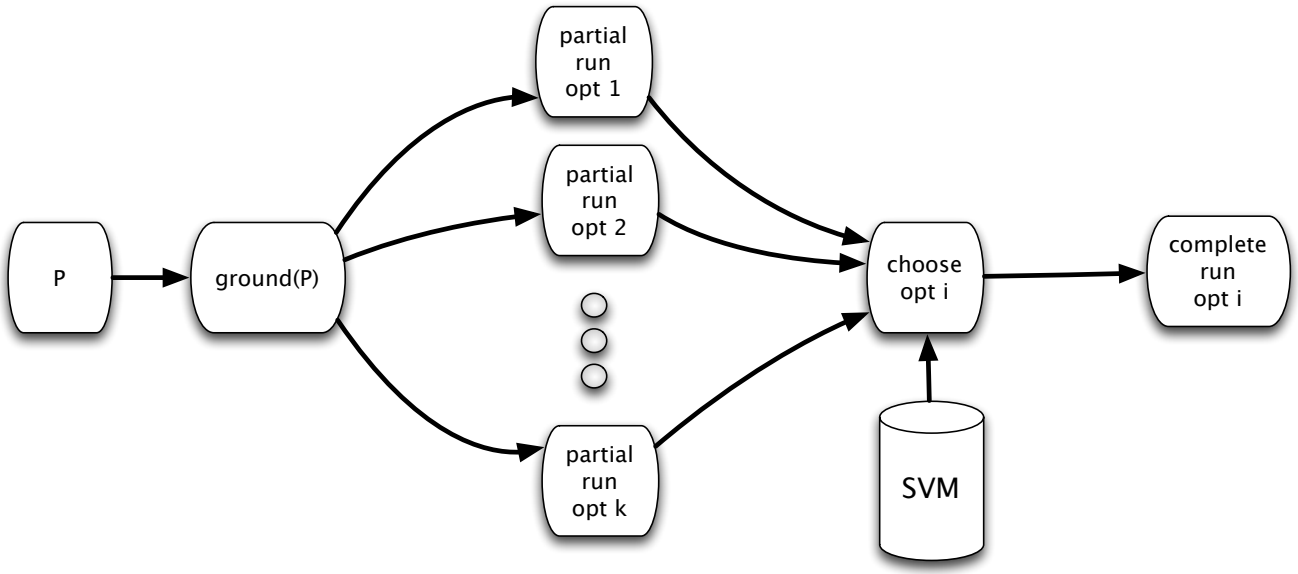
- Map-Reduce for grounding
- parallelization of Datalog (definite programs) [Afrati et al. 2011]
- computation of the Well-Founded model (normal programs) [Faber et al. 2014]
- computation of answer sets

Portfolio and multi-engine ASP-solvers

An orthogonal form of parallelism consists in exploiting multiple solvers

- possibly tuned by selecting different configuration options, heuristics,...
- the solvers can be timeouted, then the most promising configuration is used for a complete run
- machine learning techniques might drive configuration options

The case of `claspfolio 2`:

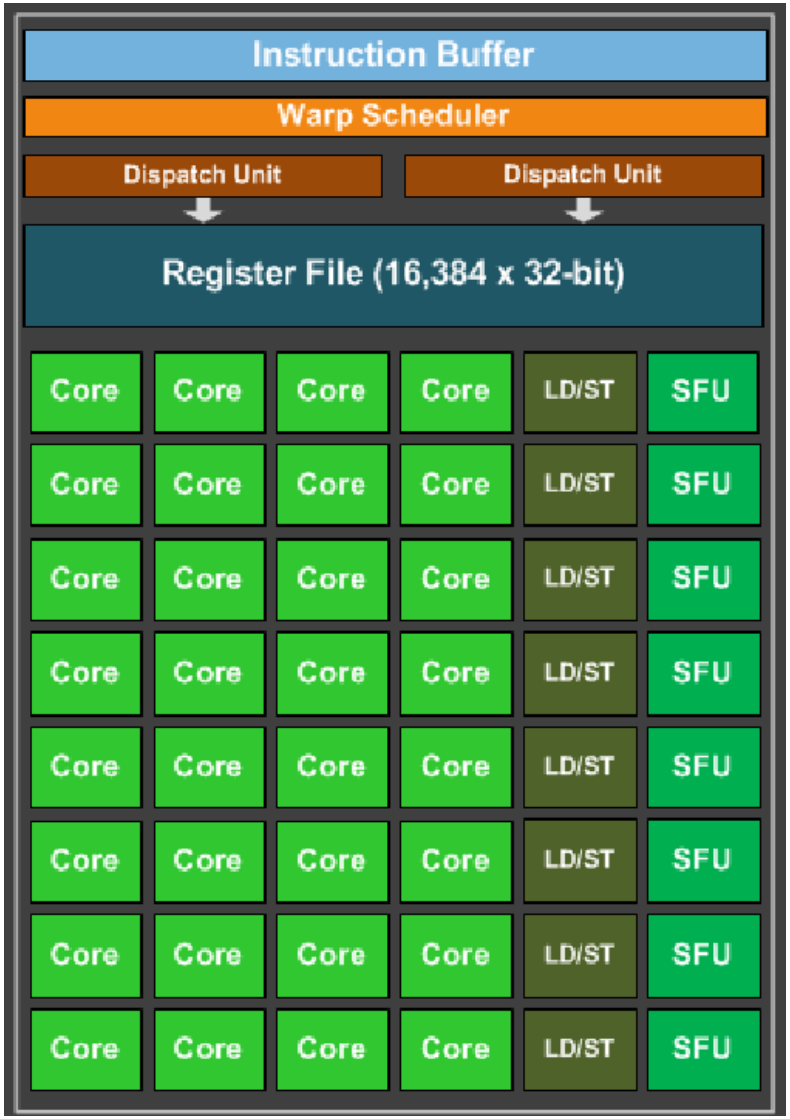


Thank you

Zoom in: A stream multiprocessor

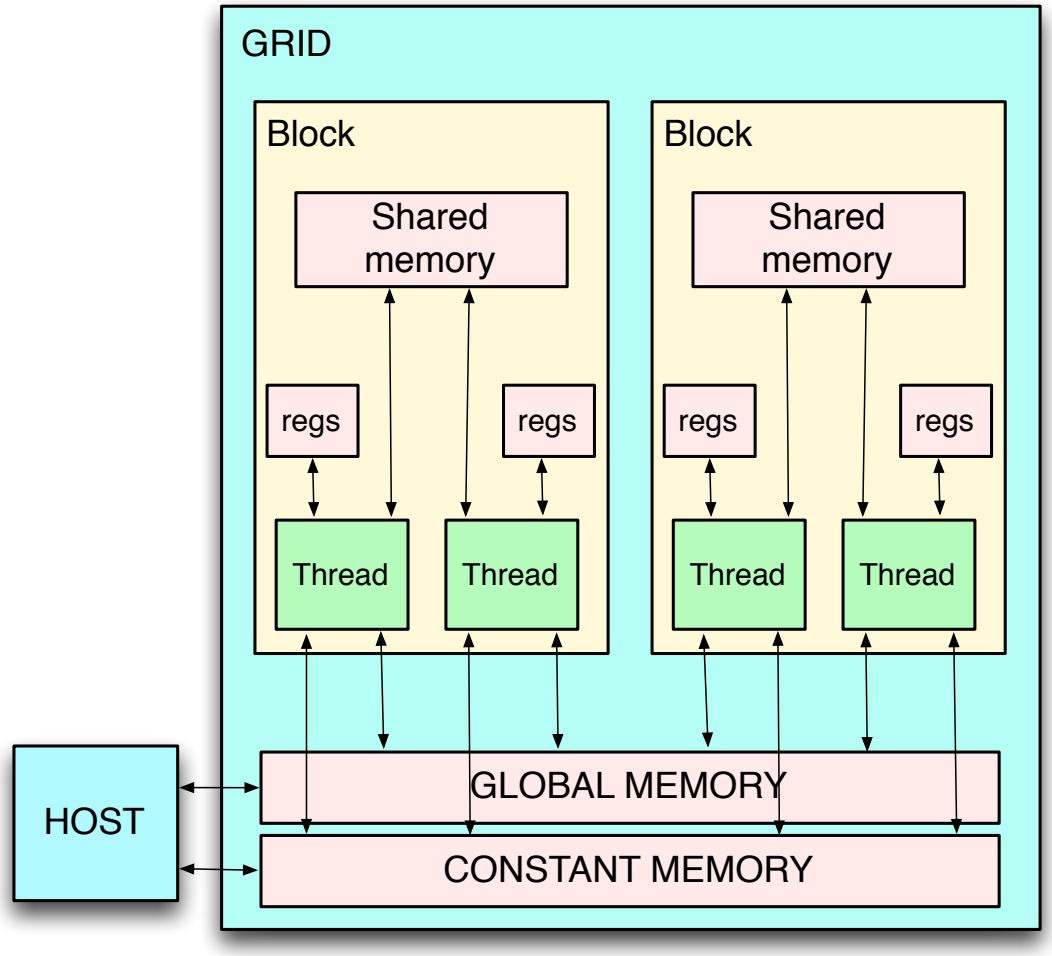
Each SMM includes

- scheduling and dispatch units
- cores
- registers
- special function units
- LD/ST units
- cache, ...



Execution model and memory hierarchy (CUDA-style)

- Each core executes a thread
  - registers
  - local memory
- warp: 32 threads
  - works in lock-step SIMT parallelism
- block: a group of threads
  - shared memory
  - synchronization support
- grid: a group of blocks
  - global memory
  - constant, texture mem.



Execution model (CUDA-style)

The computation can proceed on the host and on the device

- The programmer writes a kernel that will be run on the device
- Each thread executes an instance of the kernel

The host instructs the device:

1. copy data, host⇒device
2. kernel call
3. kernel execution on GPU
4. retrieve results, host⇐device