Scheduling in Parallel Logic Programming Systems

- Logic programming systems have been implemented on different platforms:
  - Vectorial machines. Ex.: Multipath (Convex 2000), MultiLog (BBN, MasPar MP1).
  - Distributed memory machines. Ex.: Opera (Transputers), Plosys (SP1), DORPP (EDS), clusters of workstations.
  - Distributed shared memory machines. Ex.: simulao utilizando o Mint e Treadmarks.
  - Shared memory machines. Ex.: &-Prolog, Aurora, Muse, Andorra-I, Parlog, Ace, DASWAM, Reform Prolog (Sequent, Sun, Pentium, SGI Origin).

Scheduling in Parallel Logic Programming Systems

- Different implementations require different scheduling algorithms.
- Systems do not necessarily follow the Prolog operational semantics.
- Implementations can be process-based or WAM-based.
- Can exploit various kinds of parallelism at data level (path, unification, reform) or control level (and, or, tabling).
- Can find the first solution or all solutions.
- Tradeoff: keep two of the following at constant time: variable creation, variable access, task switching.

Parallel Logic Programming Systems

- Multipath and Multilog: compute solutions using breadth-first search. Solutions S for the first goal found by backtracking.
  Continuation is computed in parallel for every solution in S.
- Scheduling problem: choose next solution path to follow.
- Very fine grained tasks.

Parallel Logic Programming Systems

- Opera, DORPP, Plosys, Aurora, Muse: compute solutions for several goal alternatives in parallel.
- Scheduling problem: choose what alternative (node) to take next.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- &-Prolog: In parallel compute multiple solutions for goals that are independent by using recomputation.
- Scheduling problem: choose what goal to execute next.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- Andorra-I: Compute *determinate* goals first and in parallel. In parallel, compute alternative solutions for the leftmost non-determinate goal, when there is no more determinate goal available.
- Scheduling problem: choose between two different kinds of work.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- Parlog: Implements a commited-choice language with operational semantics different from Prolog.
- Scheduling problem: choose next goal to execute.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- ACE: combines exploitation of independent and-parallelism and or-parallelism using stack-copying, and recomputation.
- Scheduling problem: choose between two different kinds of work.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- DASWAM: computes dependent and independent-and parallelism by using a producer-consumer algorithm.
- Scheduling problem: what goal to select next.
- Depending on the benchmark, fine, medium or coarse grained tasks.

Parallel Logic Programming Systems

- Reform Prolog: computes solutions using data parallelism on recursion.
- Scheduling problem: what goal to select next.
- Fine grained tasks.

Scheduling in Parallel Logic Programming Systems

- Finish parallel execution as soon as possible by allocating processors to the **best** tasks.
  - Keep algorithms simple to avoid too much overhead.
  - Keep number of task switches to a minimum.
  - Select work according to constraints given by the system or by the user (e.g.: communication latency or CPU cost).
  - Maintain Prolog operational semantics?
  - When possible do not do more work than the sequential execution.
  - Use dynamic, static or semi-dynamic algorithms.
  - Granularity analysis can help.

Scheduling Or-Parallelism

In general:

- Tree divided into public and private areas.
- Bottom most versus topmost dispatching.
- Deal with speculative work.
- Deal with side-effects.
- Deal with cuts and commits.
- Cost of moving on the tree.
- Different decisions to take next alternative/node.
- Dynamic work distribution.

Muse

- 4 WAM stacks: CP, Env, Heap, Trail.
- Shared and nonshared CP nodes.
- Stacks copied among workers.
- To reduce overhead of copying, workers have identical local address spaces (implemented with `mmap`) and each worker copies only the difference between its state and the other worker's state (incremental copying).
- Each worker is capable of accessing other workers' stacks.

Muse

Basic scheduling algorithm:

When a worker finishes a task, it attempts to get the nearest piece of available work on the current branch. If none exists, it attempts to select a busy worker with excess local work for sharing. If none exists, it becomes idle and stays at a suitable position on the tree.

Muse

- Matching workers:
  - find a set of busy workers, within its subtree, that are not closer to any other idle worker. Then select the one with maximum load.
  - If no worker with maximum load, find a set of busy workers outside its subtree that are not closer to any other idle worker.
  - Backtrack to the outside tree.
  - repeat first step.
- Most recent issue: dealing with cuts and speculative work: try to position workers at the leftmost part of the tree.

Muse

- Matching workers within the current subtree:
  - For each worker: local bitmaps, *busy-set* and *idle-set*. Busy-set is a list of busy workers on an idle worker's subtree. Idle-set is a list of idle workers on an idle worker's subtree.
  - busy-set and idle-set taken from global bitmaps *workersbitmap* and *idlemap*.
  - Worker removes from its busy-set busy workers that are in a subtree of any idle worker in its idle-set. Positions of idle-set workers taken from *currentnode* registers.
  - Select from the new busy-set the worker that has greater *load counter*.

Muse

put here figure of page 8 from: Full Prolog and Scheduling Or-Parallelism in Muse

Muse

- Avoiding competition to request work from the same worker: *invisibleworkersmap*. When an idle worker Q selects a worker P to grab work from, it sets two corresponding bits in the bitmap. Other workers when backtracking do not see P and Q.
- Other data structures: visible-busy-set and visible-idle-set.

Muse

put here figure of page 10 from: Full Prolog and Scheduling Or-Parallelism in Muse

Muse

- Idle workers: using heuristics to position idle workers at suitable possitions in the tree.
- Idle workers will stay at nodes that allow newly generated work to be shared with low overhead.
- Best heuristic in practice: blocking idle workers from backtracking when they are in any idle worker's subtree and move topmost idle workers to a common node owned by busy workers.

Aurora

- 4 stacks: CP, Env, Heap, Trail.
- Special data structure to store conditional bindings: Binding Array.
- Binding Array local to each worker.
- Shared stacks among workers.

Aurora

- Original scheduling approach: dispatching on topmost and sharing at most one parallel node on each branch at a time (Argonne, Wavefront, Manchester).
- No voluntary suspension.
- Advantage: minimises shared region, increases task size.
- Disadvantage: more overhead to move on the tree.

Aurora

- Better approach in practice: dispatching on bottomost. Can keep more than one live, shareable node on each branch according to heuristics based on number of idle workers and amount of work to be done (Bristol and Dharma schedulers).
- Voluntary suspension.

Aurora: Bristol scheduler

- Strategy: hybrid, *start random, finish work* for mandatory work and *start left, seek better* for speculative work.
- Start random: work taken from bottom of a branch selected evenly in a pseudo-random fashion.
- Finish work: complete the task before seeking for new work.
- Start left: choose the leftmost available work in the tree.
- Seek better work: when a worker is working on a speculative work, it will actively seek for mandatory or less speculative work. Concentrate resources on the least speculative available work.

Aurora: Bristol scheduler

- Find lefmost work: visit search tree, node by node, from left to right.
- Disadvantage: expensive due to synchronisation to access nodes.
- Idle and busy workers use an interrupt flag vector to store computation states. Whenever an idle worker finds a suitable piece of work, it sets the corresponding flag.
- While an idle worker looks for work registers interest on the nodes on its way to avoid competition.

Aurora: engine-scheduler interface

put here diagrams 1 e 2 from: The Engine-Scheduler Interface in the Aurora Or-Parallel Prolog System

Aurora: engine-scheduler interface

put here diagrams 1 e 2 from: The Engine-Scheduler Interface in the Aurora Or-Parallel Prolog System

Aurora: Dharma Scheduler

- Main objective: to deal with speculative work.
- Considers the whole tree as speculative.
- *tip* to represent each branch in the tree.
- Global double linked chain: *active chain* to represent left-to-right order of branches.
- When a new branch is added to the tree, a new tip is added to the active chain.
- When a branch is removed, the corresponding tip is removed.
- No locks are necessary and search for leftmost work is faster.

Dharma Scheduler

put diagram 8 from ILPS'93, page 407

Dharma Scheduler

put diagram 9 from ILPS'93, page 407

References

[1] Andrzej Ciepielewsky, Khayri A. M. Ali, and Prasenjit Biswas. Execution Models and Parallel Implementations of Prolog. unpublished, June 1993.

[2] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Press, 1994. http://www.cs.nmsu.edu/lldap/.

[3] Gopal Gupta, Khayri Ali, Mats Carlsson, and Manuel Hermenegildo. Parallel Logic Programming: A Survey. Research Report, ftp://pub.cs.nmsu.edu/pub/papers/lp/survey.ps.Z.