A Tutorial on

Parallel and Distributed Model Checking

Orna Grumberg

Computer Science Department

Technion, Israel

Aspects of parallelism

- Why to parallelize – gain memory or time
  - For model checking: usually memory
- Special purpose hardware or network of workstations
  - Networks of workstations
- Distributed or shared memory
  - Distributed memory with message passing

Parallel and distributed algorithms were developed for

- Explicit state methods
  - reachability and model construction
  - LTL model checking
  - model checking for alternation-free μ-calculus
- BDD-based methods
  - reachability and generation of counter example
  - model checking for full μ-calculus
- Operations on BDDs
- SAT solvers
- Timed and probabilistic model checking

Elements of distributed algorithms

- Partitioning the work among the processes
- Dynamic or static load balance to maintain balanced use of memory
- Maintaining a good proportion between computation at each process and communication
- Distributed or centralized termination detection

Reachability analysis (BDD- based)

References

"A scalable parallel algorithm for reachability analysis of very large circuits",

Heyman, Geist, Grumberg, Schuster, (CAV'00)

Also:

- Cabodi, Camurati, and Quer, 1999
- Narayan, Jain, Isles, Brayton, and Sangiovanni-Vincentalli, 1997

Reachability analysis

Goal:

Given a system (program or circuit) to compute the set of reachable states from the set of initial states

Commonly done by Depth First Search (DFS) or Breadth First search (BFS)

Symbolic (BDD-based) model checking

- **BDD** – a data structure for representing Boolean functions that is often concise in space
- BDDs are particularly suitable for representing and manipulating **sets**
- **Symbolic model checking algorithms**
  - hold the transition relation and the computed sets of states as BDDs
  - Apply **set operations** in order to compute the model checking steps

Sequential reachability by BFS

```
Reachable = new = InitialStates
While  (new $\neq \phi$) {
    next = successors(new)
    new = next \ reachable
    reachable = reachable $\cup$ new
}
```

The distributed algorithm

- The state space is partitioned into **slices**
- Each slice is **owned** by one process
- Each process runs **BFS** on its slice
- When **non-owned** states are discovered they are **sent** to the process that owns them

Goal: reducing space requirements

(and possibly time)

The distributed algorithm (cont.)

The initial sequential stage

- BFS is performed by a single process until some threshold is reached
- The state space is **sliced** into k slices.
  Each slice is represented by a **window function**.
- Each process is informed of:

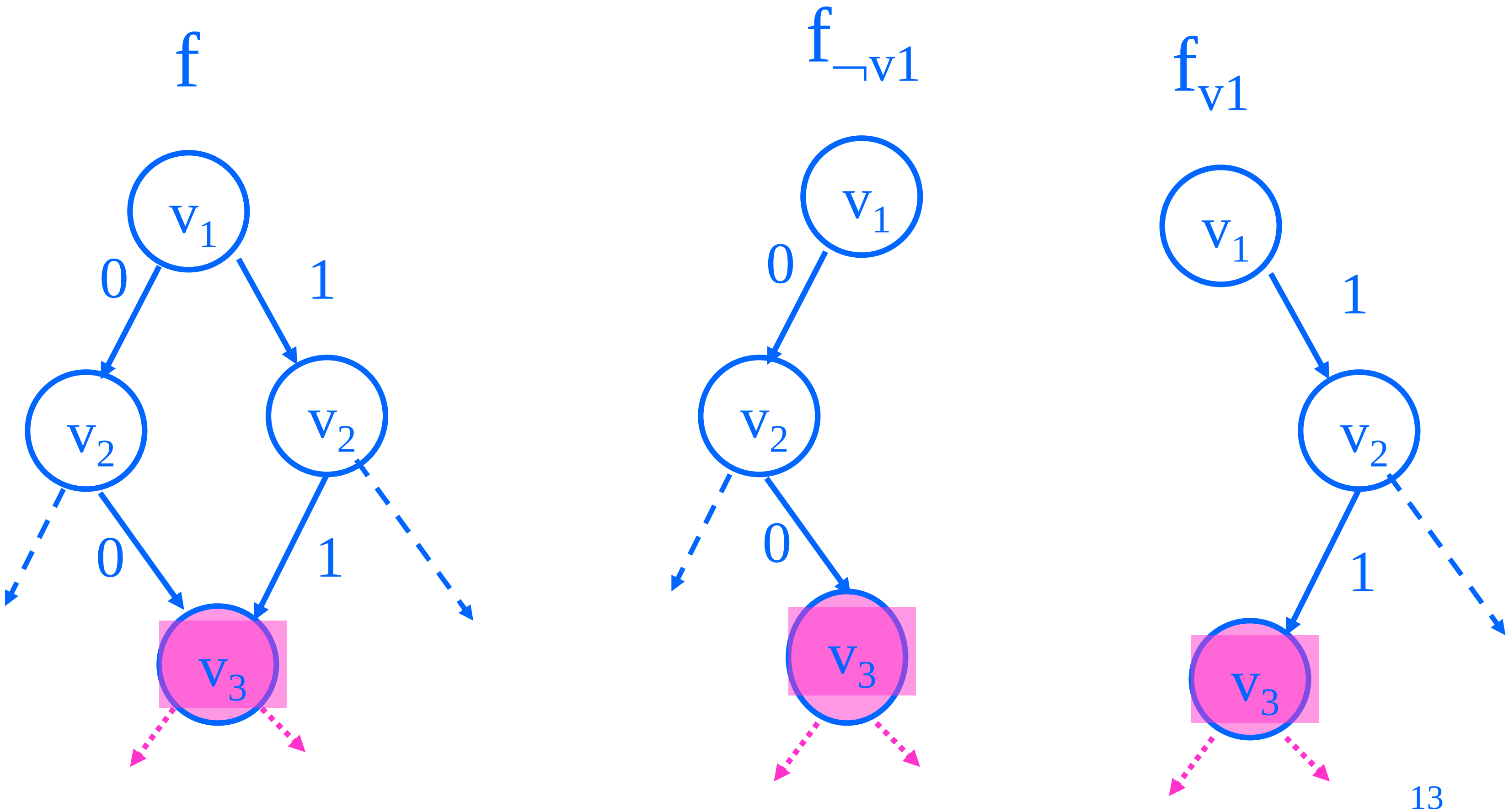
  - The set of **window functions $W_1,\ldots,W_k$**
  - Its own slice of **reachable**
  - Its own slice of **new**

Elements of distributed symbolic algorithm

- **Slicing algorithm** that partitions the set of states among the processes
- **Load balance** algorithm that keeps these sets similar in size during the execution
- **Distributed termination detection**
- Compact **BDD representation** that can be **transferred** between processes and allows **different variable orders**

Slicing:

When slicing a BDD we loose the sharing that causes BDD to be a compact representation



Slicing (cont.)

- We choose a variable **v** and partition a Boolean function **f** to $\mathbf{f}_{\mathbf{v=1}}$ and $\mathbf{f}_{\mathbf{v=0}}$.
- The chosen v has **minimal cost** that guarantees:
  - The size of each slice is below a threshold.
    I.e., the partition is not trivial
    (no $|f_1| \approx |f_2| \approx |f|$ or $|f_2| \approx 0$)
  - The duplication is kept as small as possible
- An **adaptive cost function** is used to keep the duplication as small as possible

Load Balance

- The initial slicing distributes the memory requirements equally among the processes.
- As more states are discovered, the memory requirements might become **unbalanced**.
- Therefore, at the end of each step in the computation of the reachable states a load balance procedure is applied.

Load Balance (Cont.)

- Process i with a small slice sends its slice to process j with a large slice.
- Process j applies the slicing procedure for k=2 and obtains two new balanced slices.
- Process j sends process i its new slice and informs all other processes of the change in windows.

The parallel stage requires a coordinator for:

Pairing processes for exchanging non-owned states

- Processes notify the coordinator of the processes they want to communicate with.
- The coordinator pairs processes that need to communicate.
- Several pairs are allowed to communicate in parallel.

Important: Data is transferred directly between the processes and not via the coordinator

Coordinator can also be used for:

- Pairing processes for load balancing
- Distributed termination detection

Experimental results

On 32 non-dedicated machines, running IBM RuleBase model checker:

- On examples for which reachability terminates with one process, adding more processes reduces memory (number of BDD nodes)
- On examples for which reachability explodes, more processes manage to compute more steps of the reachability algorithm

- Communication was not a bottleneck
- Time requirements were better in some examples and worse in others
  - better – because BDDs are smaller
  - worse – overhead and lack of optimizations for improving time

Future work

- Improve slicing
- Exploit different orderings in different processes
- Adapt the algorithm to dynamic networks
- Adapt the algorithm for hundreds and thousands of parallel machines

Model checking safety properties

- Checking **AGp** can be performed by
  - Computing the set of **reachable** states
  - For each state checking whether it satisfies **p**
- If a state which satisfies ¬p is found, a **counter example** – a path leading to the error state - is produced
- Checking other safety properties can also be reduced to reachability

Back to reachability (explicit state)

References

Explicit state reachability:

- “Distributed-Memory Model Checking with SPIN”, Lerda and Sisto, 1999

Also:

- Caselli, Conte, Marenzomi, 1995
- Stern, Dill 1997
- Garavel, Mateescu, Smarandache, 2001

LTL model checking:

- “Distributed LTL Model-Checking in SPIN”, Barnat, Brim, and Stribrna, 2001

Sequential reachability

- States are kept in a hash table
- Reachability is done using a DFS algorithm

Distributed Reachability

- The state space is **statically** partitioned
- When a process encounters a state that does not belong to it - the state is sent to the owner
- Received states are kept in a FIFO queue
- **Verification ends** when all **processes are idle** and all **queues are empty**

Choosing the partition function

- Must depend only on the state
- Should divide the state space evenly
- Should minimize cross-transitions

- First solution - partition the space of the hash function
  - ☒ cannot be implemented on a heterogeneous network
  - ☑ even distribution, but not necessarily of the **reachable** states
  - ☒ does not minimize cross transitions

A better partition function for asynchronous programs (like in SPIN)

A **global state s** consists of the **local states s_i** of each concurrent sub-program

- Choose a specific sub-program $prog_i$
- Define the **partition function** according to the **value of the local state s_i** of sub-program $prog_i$

Since a transition generally involves one or two sub-programs, this partition

  ☑minimizes cross-transitions

  ☑distributes the state-space evenly

LTL model checking with Büchi automata

(explicit state)

LTL model checking with Büchi automata

- A Büchi automaton is a finite automaton on **infinite words.**
- An infinite word is **accepted** if the automaton, when running on this word, visits an accepting state **infinitely often.**
- Every **LTL** formula can be translated into a Büchi automaton that accepts **exactly all** infinite paths that satisfy the formula.

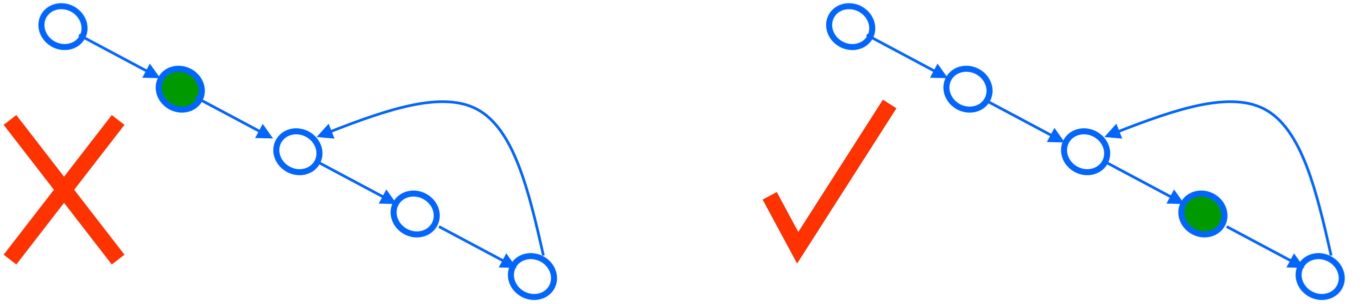
Checking M |= φ for an LTL formula φ

In order to verify a property φ, an automaton $A_{\neg\varphi}$ is built.

- $A_{\neg\varphi}$ contains all behaviors that satisfy ¬φ.
- M x $A_{\neg\varphi}$ contains all the behaviors of M that do not satisfy φ.
- M |= φ iff M x $A_{\neg\varphi}$ is empty.

Checking for (non)emptiness

- Looking for a reachable loop that contains an accepting state
- Tarjan’s algorithm, O(|Q| + |T|)



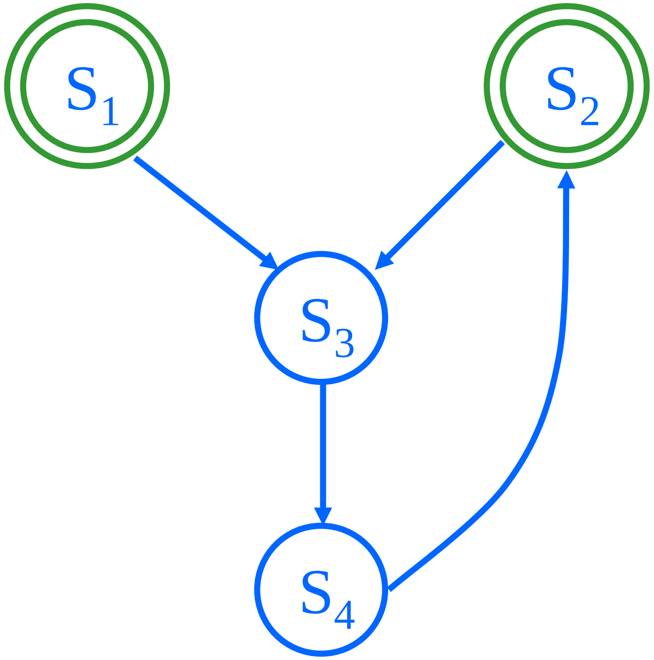
Nested DFS Algorithm

- Two DFS searches are interleaved
  - The first looks for an **accepting state**
  - The second looks for a **cycle back** to this state
- When the first DFS backtracks from an accepting state it starts the second (nested) DFS

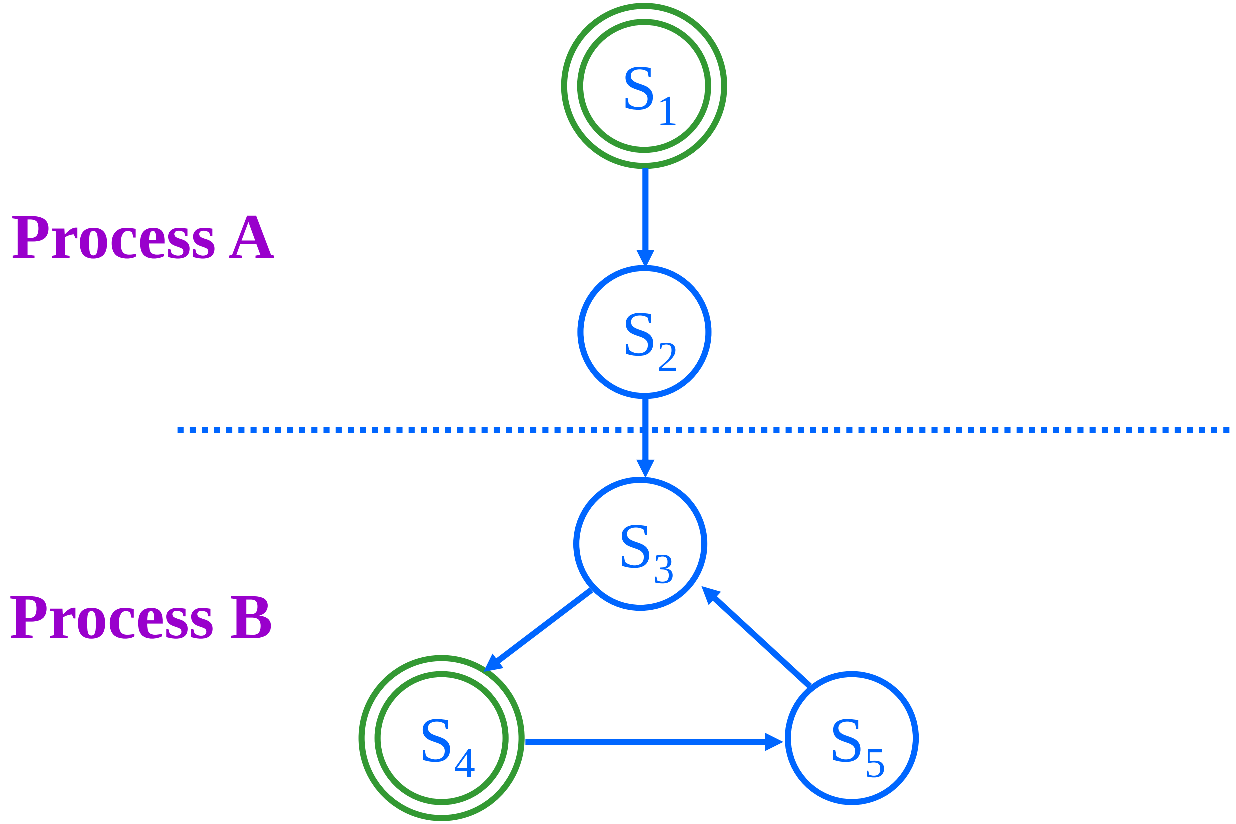
- The second DFS looks for a loop back to the accepting state
- When the second DFS is done (without success) the first DFS resumes
- **Each DFS goes through every reachable state only once!**

Why nested-DFS won’t work in parallel

Relative speed determines if a cycle is found



- The order matters



A nested DFS should start from s iff all accepting states below s have finished their nested DFS

Inefficient solution

- Holding for each state the list of NDFSs it participated in
  - Requires too much space
  - Allows each state to be traversed more than once for each of the two DFSs

Main characteristics of the distributed algorithm

- Dependency graph, containing only **accepting states** and **border states**, is used to preserve **limited amount of information**
- Each process holds its own dependency graph
- NDFS starts from a state only after all its successors are search by DFS and NDFS
- NDFS is **not performed in parallel** with another NDFS

Experimental results

- Preliminary
- 9 workstations interconnected by Ethernet
- Implemented within SPIN and compared to standard, sequential SPIN
- Could apply LTL model checking to larger problems

Future work

- Improve the partition function
- Increase the level of parallelism by allowing NDFSs to work in parallel under certain conditions

SAT-based model checking

State explosion problem in model checking

- Systems are extremely large
- State of the art symbolic model checking can handle medium to small size systems effectively:
  a few hundreds Boolean variables

Other solutions for the state explosion problem are needed.

SAT-based model checking

- Translates the model and the specification to a propositional formula
- Uses efficient tools for solving the satisfiability problem

Since the satisfiability problem is **NP-complete**, SAT solvers are based on **heuristics**.

SAT tools

- Using heuristics, SAT tools can solve very large problems fast.
- They can handle systems with 1000 variables that create formulas with a few thousands of variables.

GRASP (Silva, Sakallah)

Prover (Stalmark)

Chaff (Malik)

Bounded model checking for checking AGp

- Unwind the model for k levels, i.e., construct all computations of length k
- If a state satisfying ¬p is encountered, then produce a counter example

The method is suitable for **falsification**, not verification

Bounded model checking with SAT

- Construct a formula f_M describing all possible computations of M of length k
- Construct a formula f_φ expressing $\varphi = EF\neg p$
- Check if $f = f_M \wedge f_\varphi$ is satisfiable

If f is satisfiable then $M \not\models AGp$

The satisfying assignment is a **counterexample**

Bounded model checking

- Can handle **LTL** formulas, when interpreted over finite paths.
- Can be used for **verification** by choosing k which is large enough so that every path of length k contains a cycle.
  - We then need to **identify cycles** using propositional formulas.
  - Using such k is often **not practical** due to the size of the model.

SAT Solvers

Main problem: time

Secondary problem: space

References

- “PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems”, Zhang, Bonancina and Hsiang, 1996
- “PaSAT – parallel SAT-checking with lemma exchange: implementation and applications”, Sinz, Blochinger, Kuchlin, 2001

- Also:
- Bohm, Speckenmeyer 1994
- Zhao, Malik, Moskewicz, Madigan, 2001

Propositional formula in Conjunctive Normal Form (CNF)

CNF consists of a conjunction of clauses.

A **clause** is a disjunction of literals.

A **literal** is either a proposition or a negation of a proposition.

$(a \vee \neg e) \wedge (c \vee b) \wedge (\neg c \vee d) \wedge (\neg c)$

A Simple Davis-Putnam Algorithm

```
Function Satisfiable (set S)
                        return boolean
  repeat /* unit propagation */
     for each unit clause L in S do
        delete (L ∨ Q) from S    /* unit subsumption */
        delete ¬L from (¬ L ∨ Q) in S  /* unit resolution */
     od
     If S is empty return TRUE
        else if null clause in S return FALSE
  until no further changes result
```

Davis-Putnam Algorithm (Cont.)

```
  /* case splitting */
  choose a literal L occurring in S
  if Satisfiable ( $S \cup \{L\}$ ) return TRUE
    else if Satisfiable ($S \cup \{\neg L\}$) return TRUE
    else return FALSE
end function
```

$(a \vee \neg e) \wedge (a \vee e) \wedge (c \vee b) \wedge (\neg c \vee d) \wedge (\neg c)$

- Unit clause: **$c=0$**
  $(a \vee \neg e) \wedge (a \vee e) \wedge (\ b)$
- Unit clause: **$b=1$**
  $(a \vee \neg e) \wedge (a \vee e)$
- Selecting splitting literal: **$a=0$**
  $(\neg e) \wedge (\ e)$ – **conflict!**
- Create conflict clause: **$(c \vee \neg b \vee a\ )$**
- Backtracking and choosing $a=1$
- Satisfying assignment: $c=0$, $b=1$, $a=1$

Points of wisdom

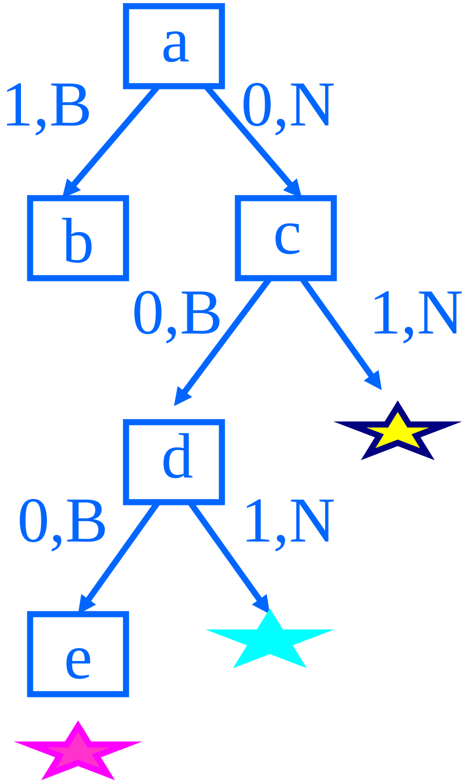
- Clever choice of the splitting literal.
- Clever **back-jumping** on unsuccessful assignments.
- Remembering unsuccessful assignments as **conflict clauses** or **lemmas**.

PSATO

A distributed implementation of SAT on network of workstations.

- The goal is to exploit their under-used computation power, especially after hours: **parallelize** and **cumulate** the work
- **Dynamic load balance** is needed since the computing power of each workstation is not known in advance (it may be shared with other programs).

Partitioning the work



★ a <0,N> c <1,N>

★ a <0,N> c <0,**N**> d <1.**N**>

★ a <0,N> c <0,**N**> d <0.**N**> e

The Master-Slave Model of PSATO

- One **master**, many **slaves**
- Communication only between master and slaves
- Master **sends jobs** **(S,P)** to slaves
  S – set of clauses, **P** – guiding path
- Each **slave** runs Davis-Putnam according to **P**
- When a **slave stops**, it sends master
  - TRUE or FALSE, if job is finished
  - guiding path, if job is interrupted

Balancing the workload

- If a slave returns **TRUE,** all slaves are **stopped**
- If it returns **FALSE,** the slave is assigned a **new path.**
- If **time expires**, the master sends halt signal to stop the current run and **collects new paths**

The new paths will be used in the next run

Achievements

- **Accumulation of work:** cumulates the results of separate runs on the same problem
- **Scalability:** more workstations result in a faster solution
- **Fault tolerance:** minimal damage by failure of one workstation or network interruption
- **No redundant work:** processes explore disjoint portions of the search space

Experimental results

- For random hard 3-SAT problems, the speedup on 20 machines was from 6 to 18.
  - Speedup is the ration between CPU time of the sequential machine and the average time over the parallel machines.
- For open quasigroup problems they managed to solve a problem on 20 machine in 35 “working days” that would otherwise require 240 days of continuous run on a single machine.

PaSAT

Can run on multi-processor computer and on a networked standard PCs

Implemented on shared memory with dynamic creation of threads

PaSAT (Cont.)

- Uses guiding paths as in PSATO for partitioning the work and for balancing it
- Holds conflict clauses learned by all tasks in a shared memory
  - implemented so that it allows concurrent access without synchronization
- Each task filters its conflict clauses and put only the “best” in the global store
- Periodically, each task integrates new clauses from the global store into its current set

Experimental results

- On a machine with 4 processors, on satisfiable SAT problems:
  obtained a speedup (time on sequential/time of parallel) of up to 3.99 without exchange of conflict clauses and even higher with the exchange

Future work

- Implement the ideas of PaSAT on distributed memory
- Extend the ideas for many machines working in parallel

THE END