Concurrent-Distributed Programming Techniques for SAT
Using DPLL-Stålmarch

Sripriya G, Alan Bundy, Alan Smaill
School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh, EH8 9AB, UK
{sripriya.g, a.bundy, a.smaill}@inf.ed.ac.uk

ABSTRACT

This paper reports our work on application of concurrent/distributed techniques to SAT. These were investigated using each of the following methods: the DPLL [5] algorithm, the dilemma rule of the Stålmarch’s algorithm [12] and a concurrent/distributed hybrid SAT solver: DPLL-Stålmarch, using a combination of the DPLL and the dilemma rule based algorithm. The prototypes have been implemented using Alice [10], an SML based language with support for distribution and concurrency. Our prototype framework allows for rapid-prototyping of and experimentation with application of various concurrent/distributed programming techniques to SAT. The emphasis is not on building an industry-standard SAT solver, but rather an investigation of the efficacy of use of these techniques for SAT at various levels of granularity.

We discuss in detail the workings of DPLL-Stålmarch. DPLL and Stålmarch are complementary methods given their depth-first and breadth-first approaches. A single threaded execution limits the possibilities of combining the two methods. We have used our prototype framework to explore the use of concurrent/distributed programming techniques for achieving cooperation between the two methods. Empirical evaluation is in progress using the metrics of time, size of search space, and number of machines.

The work reported is part of an ongoing project investigating the use of concurrent/distributed programming techniques for theorem proving.

KEYWORDS: SAT, Concurrency, Distribution, DPLL, Stålmarch, Combination of solvers, Functional programming

1. INTRODUCTION

There is intense interest in efficient SAT solvers driven by the theoretical importance of SAT and also the practical significance of SAT solvers. Many interesting and important problems from a variety of domains can be encoded as SAT problems. This has fueled the increased interest and has also provided bigger and more complex problems for SAT solvers.

With the almost saturated capacities of single processor computers and the increasing accessibility of architectures and infrastructure that facilitate concurrent/distributed programming paradigms, it is becoming imperative to investigate novel ways of using these concurrency and distribution technologies to tackle hard problems. Furthermore, these approaches can give scope to go beyond raw parallelisation approaches and allow for tapping latent distribution and collaborative problem solving opportunities present in various tasks that computers are being used to solve.

Thus, the scope provided by the application of concurrent-distributed programming techniques to build faster and smarter SAT solvers merits serious investigation. In the context of the work described in this paper, the emphasis is not on building an industry-standard SAT solver, but rather an investigation of the efficacy of use of concurrent-distributed programming techniques for SAT at various levels of granularity. We discuss our experiments on applying these techniques to SAT solvers based on: the DPLL [5] algorithm, the dilemma rule of the Stålmarch’s algorithm [12] and a hybrid-solver using a combination of the DPLL and the dilemma rule based algorithm.

Our prototypes have been developed in Alice[10], a func-
tional programming language with support for concurrency and distribution. Alice’s feature-rich, light-weight thread support in a functional setting has enabled our prototype framework to serve as an exploratory workbench for rapid prototyping of and experimentation with application of techniques such as the following to SAT: simultaneous exploration of choices, picking the fastest-returning one, producer-consumer pattern, message-passing techniques.

2. MOTIVATION

2.1. Why do we need concurrent-distributed-parallel techniques for SAT?

The last decade has seen significant improvements in SAT technology. These include development of efficient data structures, heuristics, non-chronological backtracking with conflict-clause learning and highly fine-tuned implementations. Most of these use the DPLL [5] as the core algorithm. Also, these efforts focus primarily on building an efficient blackbox SAT solver. Alternative approaches can be to: make a SAT solver receptive to external information which can be in the form of clauses or other hints like structural information [8]; use of multiple solvers; combining complementary solver methods [1], [2].

A single threaded execution limits the possibilities of combining various solvers and the interaction between heuristics and the main solver. We hypothesize that the application of asynchronous forms of computation and concurrent/distributed programming techniques to SAT solvers is a promising research direction to pursue. Some arguments for the same are:

- Open up fresh approaches that hitherto were not possible
- Performance gains via application of these techniques to existing SAT solver methods
- Give insights into the workings of existing SAT solvers, in particular, the interplay between heuristics and the main solver
- Optimal utilisation of multicore and distributed technology platforms
- Effective utilisation of (idle) resources
- To explore effective combinations of different solving techniques

2.2. Need for a framework for rapid prototyping and experimentation

Many of the industry-standard SAT solvers involve fine-tuning of various parameters, cache performance and the hardware that the solver is being run on. These SAT solvers are mostly written in C and applying concurrent/distributed programming techniques at a fine level of granularity to these systems is a complicated exercise and can often compromise the fine-tuning that makes them so efficient in the first place. It has been widely observed that functional programming is very well suited for concurrent programming. Also, a light-weight thread mechanism and rapid-prototyping facilities allows for easier and richer experimentation. Comparisons can be made with sequential solvers developed in the same framework. The results thus gleaned can be used to port the distributed programming abstractions to other industry standard solvers as well, using their own infrastructure for distribution.

3. BACKGROUND

3.1. Stålmarck’s algorithm

Stålmarck’s algorithm [12] is used in the tools from Prover technology (www.prover.com) and also in the experimental system HeerHugo [9]. While the DPLL is based on a depth-first approach, Stålmarck’s algorithm can be interpreted as a breadth-first approach over all possible trees in increasing depth, with several enhancements.

It has been observed that the success of the Stålmarck’s prover should be solely attributed to the dilemma rule, since its other aspect (namely the handling of triplets), is a slightly enhanced variant of unit resolution [9]. For the purpose of this paper, we primarily use the dilemma rule of Stålmarck’s algorithm and any references to Stålmarck’s algorithm concerns the dilemma rule, also referred to as the branch-and-merge rule in the literature. Another key difference in our implementation is the following: Stålmarck’s algorithm works on triplets and uses a set of so-called simple rules along with the dilemma rule system. Our algorithm operates on formulas in conjunctive normal form (CNF) and incorporates many logical rules that are more or less common in theorem proving, like unit-propagation. The branch-and-merge rule is depicted in the figure below.

In the second branch, the assumption p is added to Φ. Exhaustive application of the simple rules gives a set of conclusions C_p. Conclusions are of the form q or ¬q. C¬p is obtained in a similar fashion. After exploring both branches, the intersection of C_p and C¬p is calculated. Each element of the intersection is a fact and is added to the formula, Φ
and the simple rules are applied to the resulting formula. The branch-merge is applied to all the propositional variables. The next levels involves nested applications i.e, repeat for all pairs, triples, ... till either SAT or UNSAT is proved.

3.2. Alice

Our framework and the prototype solvers have been developed in Alice, a functional programming language based on SML, with support for concurrency and distribution. The lightweight concurrency-distribution features allow for rapid-prototyping and experimentation. This has been extremely valuable, given the experimental nature of this prototype effort, where the objective has been to investigate the concurrency-distribution techniques that can be gainfully employed and if it brings any performance gains with respect to comparable sequential versions. Some of the features of Alice relevant to the requirements for our framework are: data-flow synchronisation, ease of implementation of message-passing mechanisms, remote-process-invocation, ability to pass rich content via the message passing mechanisms and lazy evaluation. Furthermore, having the features as part of the language, rather than as a library, enables seamless integration and inter-operability between the language features and the distributed facilities. Alice allows for effective utilisation of multi-core architectures. In our current version, we have targeted distributed systems, i.e, a network of computers. We are working on extending this to target multicore architectures.

4. IMPLEMENTATION:OVERVIEW

A SAT solver has been developed from scratch using Alice [10] as the implementation language. The DPLL and Stålmarck algorithms have been implemented. Concurrency techniques have been employed within both the DPLL and Stålmarck solvers. Furthermore, a hybrid solver DPLL-Stålmarck, using both DPLL and Stålmarck methods has been implemented and it employs various concurrent and distributed programming techniques.

A literal is represented by an integer, the positive and negative integers representing the variable q and its negation ̄q respectively. A clause is represented by a set of literals. A problem is represented by a set of clauses. Other data structures like BitArray representations were considered, but, with a view towards avoiding use of references and imperative patterns, these options were not pursued. Also, use of reference-based data structures proved to be a significant overhead for implementing the concurrent techniques because of issues relating to manipulation of state/memory. A basic heuristic has been applied at choice-points: maximally occurring variable.

5. CONCURRENT/DISTRIBUTED TECHNIQUES USED

5.1. Concurrent DPLL

This solver uses the standard DPLL algorithm with the following improvisation. At each choice-point, two threads are spawned asynchronously to explore the respective subtrees. The thread that comes back first with a satisfiable assignment terminates the other thread. Furthermore, termination of a thread terminates all the sub-threads spawned under it. Bookkeeping of the number of live threads at any given time takes care of the potentially exponential number of thread-spawns. The code outline for this solver is given below. Empirical evaluation is in progress. Preliminary results show performance gains in cases where the satisfiable assignment is in a shallow level on one of the branches and exploration of the other branch takes very long. Our approach to assess the gains made by this feature has been to compare the performance of: DPLL, DPLL with orders-flipped at choice-points and concurrent-DPLL.

5.2. Distribution within Stålmarck

The key insight for our work has been the fact that in the repeated applications of the branch-merge rule of Stålmarck’s algorithm, as described in §3.1., each run is independent and can be pursued by an independent process/agent. This gives the freedom of allowing these agents to run on many different workstations without any dependencies on the state of the other processes and without creating any bottlenecks for other processes using the agent’s results. We refer to this agent as the stalmarck agent service and it is described in detail below. The facts that are derived are applicable to the entire formula. The facts that are derived are posted to Figure 1. Branch-merge Rule
fun takeFastestAndKillOther(t1,r1) (t2,r2) = case (Future.awaitEither(r1,r2) ) of FST(Sat(_)) ⇒ (Thread.terminate(t2);r1)| FST(Unsat) ⇒ r2 | SND(Sat(_)) ⇒ (Thread.terminate(t1);r2)| SND(Unsat) ⇒ r1

fun doDistributed_DPLL (prob) : result = let
  fun solveAssign(prob,lit): result = let
    val rProb = doAllUnitCl (doAllPureLit (remTauts (prob,lit)))
in  case testProb(rProb) of Sat sat_assign ⇒ Sat sat_assign |Unsat ⇒ Unsat |UNKNOWN ⇒ branch (rProb, pickBranchingLit (rProb))
and
  branch ( prob, lit) : result = let
    ( * r1, r2: _future * )
    val (t1,r1)= spawnThread (solveAssign (prob, lit) );
    val (t2,r2)= spawnThread (solveAssign (prob, ˜lit));
in  do takeFastestAndKillOther(t1,r1) (t2,r2)
end
in
  solveAssign prob
end

Figure 2. Concurrent-DPLL

a pre-defined location and/or to the subscribers to the stalmarck agent’s service, which is explained below. So, another solver can also use the facts derived by these individual agents as can the stalmarck agents themselves.

5.3. Stålmarck agents as services

A single stalmarck agent can be described as a service that is running on a remote machine and whose functions can be invoked remotely. A work stream is created using an Alice channel, say C-s. This holds the combinations to be processed. Another stream is created where the problem gets posted, say P-s. A third stream is created where the agents post their deductions, say D-s. The stalmarck agent services are bound to these three channels at the time of creation. Lazy evaluation has been used to engineer the facilities of waiting. Once triggered, the doWork procedure of the stalmarck agent service works as follows:

- Waits on a particular location, P-s, where the problem will appear
- Once a problem appears, the service will proceed to the next step, to fetch a combination, if there is no combination it will wait.

5.4. Producer-consumer pattern, Resource-management

The stalmarck agent service implementation can be viewed as an instance of the producer-consumer abstraction. Once triggered, the doWork function of the stalmarck agent service, acts as a consumer and picks a combination from C-s, the producer. It works on it and upon finishing the work, posts the results to D-s and goes to fetch another combination from the stream C-s. This allows for enforcing resource-management techniques as the user can specify the number of stalmarck agents depending on the computational resources available. For example, if the solver is deployed in a network of workstations, then the number of stalmarck agents can be adjusted to optimally utilise the available number of idle workstations.

6. DPLL-STÅLMARCK

6.1. Why combine DPLL and Stålmarck?

It has been widely observed that combining DPLL and Stålmarck’s methods is a very promising research direction, given their complementary approaches of depth-first and breadth-first respectively [1], [2]. However, attempts to implement this have suffered from restrictions imposed by a sequential system, as evidenced in the work in [1], where stalmarck’s method first runs for a predetermined period of time before another theorem proving method is used.

6.2. Implementation: DPLL-Stålmarck

Our framework allows for a far more flexible form of orchestration and interaction. We have developed a prototype of a hybrid solver with a co-operative orchestration of the DPLL and Stålmarck methods. The Stålmarck’s method does not need to be run for a predetermined period of time, but can be running concurrently and as independent agents running on a huge array of machines. Our hybrid solver
does not involve any pre-processing or judgements on when to apply DPLL or Stålmarck. The DPLL and Stålmarck processes work concurrently on the problem as follows: The stalmarck agent services run in the background and act as information-providing agents to the central DPLL algorithm/process. Thus, the DPLL process does not suffer from any bottlenecks caused by the stalmarck agents. We explain below our implementation of the hybrid solver, DPLL-Stålmarck developed using the Alice framework.

### 6.2.1. Boot strapping

As part of the boot strapping process for the DPLL-Stålmarck solver, the following actions are performed:

- A `dpllInbox` is created and this is subscribed to the stalmarck agent services. This is a persistent storage, for a given run of the solver.

- A user specified number of stalmarck agent services are triggered, say, $SA_1, SA_2, ...SA_n$. The `doWork` functions are triggered via the stalmarck agents services, which are already running on remote hosts.

- The problem is posted to the pre-defined location, $P-s$.

- All the combinations that need to be processed are put onto the work stream, $C-s$.

### 6.2.2. How DPLL uses Stålmarck?

The DPLL-Stålmarck uses the DPLL algorithm as the core, with the following enhancement. As explained above, the `dpllInbox` is subscribed to the stalmarck agent services. So, it will be notified of all deductions produced by each of the services. At each case split on a literal, the information from the `dpllInbox` is added to the problem. This can result in rapid pruning of search spaces in some cases. The process interaction overview and the code-outline are given below.

### 6.2.3. Advantage of the framework

The code-outline 4 illustrates the advantages of the framework and the ease of implementation it offers. The stalmarck agent service can be looked at as a plug-in that works as an information-providing service. Thus, this architecture can be extended to allow for other information providing agents, provided the information is valid at all stages of the DPLL algorithm.

```haskell
fun doDPLLStalmarck (prob, nAgents): result =
let
  do create_dpllInbox (* dpllInbox: Persistent,shared resource*)
  do boot_stalmarck nAgents dpllInbox
fun solveAssign(prob,lit): result =
let
  (USE OF EXTERNAL INFORMATION)
  val nProb = checkInbox (prob,lit)
  val rProb = doAllUnitCl (doAllPureLit (remTauts (nProb,lit)))
in
  case testProb(rProb) of
  Sat sat_assign => Sat sat_assign
  |Unsat => Unsat
  |UNKNOWN =>
  branch(rProb, pickBranchingLit (rProb))
and
  branch ( prob, lit) : result =
  case solveAssign (prob, lit) of
  Sat => Sat sat_assign
  |Unsat => solveAssign (prob, ¬lit)
in
  solveAssign prob
end
```

**Figure 4. DPLL-Stålmarck**

### 7. EVALUATION

As has been observed in the literature, parallel/distributed SAT solvers are hard to evaluate [14] because of extreme variations in performance and algorithmic complexity analysis not being possible. So, one has to rely on empirical data. Empirical evaluation of our prototype systems is in progress. The metrics for evaluation are time and the size of the search space that has been explored. In addition, for the DPLL-Stålmarck solver, an additional parameter is also used for evaluation: number of Stålmarck agents.
Preliminary results have been promising and in some cases show speedups for the distributed DPLL and the hybrid solver: DPLL-Stålmarck and no significant overhead have been observed in the distributed implementations considering the average case scenarios. More detailed evaluation is in progress using standard SAT benchmarks with a view towards addressing aspects of hard instances and clause-variable ratio criteria.

8. DISCUSSION

8.1. Irregular search spaces

A DPLL solver can get possibly stuck in a fruitless search branch though there might be a satisfactory assignment at a shallow level on the other branch. It has been identified that even for some relatively easy instances certain orders of search may take the algorithm into parts of the search space that do not produce useful conflict clauses, leaving it floundering. Restarts were proposed in [7] as an approach to dealing with high variance in running times over similar instances. Spawning threads asynchronously is one way of effectively addressing this issue. We have done preliminary evaluation of our concurrent DPLL solver §5.1. this aspect by having the solver flip the order of case-splits and comparing the results between the two solvers.

Another point worth observing is that in our case the information provided by the Stålmarck agents can help both in SAT and UNSAT. Because, at each case-split the information provided is used to prune the search space, where possible, thus helping more towards the UNSAT case. Also, the deductions are applied to the problem itself, thus reducing the work, contributing towards the SAT case as well.

8.2. Granularity

Opportunities of application of concurrent/distribution programming techniques are present at different levels of granularity in the context of SAT. In our work, we have investigated both fine and coarse levels of granularity. The distributed-DPLL and distributed-Stålmarck is a case of use of these techniques at a fine-level granularity and the DPLL-Stålmarck is a case of combination of coarse and fine level granularities. The external information provided can be viewed as a coarse-level granularity distribution.

8.3. Combination of approaches

With the hybrid solver DPLL-Stålmarck, the hybrid approach has the advantages of a combination of depth-first search(DPLL) and breadth-first search(Stålmarck) approaches to SAT solving. Such interleaving of multiple approaches can be further refined by using heuristics to decide what-to and when-to interleave.

8.4. Use of external information

The dynamic learning process of modern SAT solvers relies on accumulated knowledge which is continuously deduced during the solving process. In the hybrid DPLL-Stålmarck implementation, the asynchronously running stålmarck agents can be viewed as information supplying agents to the DPLL process which can use the information to possibly prune its search space. This is in contrast to the other similar work on distributed SAT solving, in PaSAT [13], wherein the additional information comes in the form of lemmas learnt as a result of conflicts within the DPLL solver.

In the case of the pure stålmarck implementation, pruning of search space is achieved as follows. The threads for each deduction-deriving process are spawned in an asynchronous manner and the derived deductions are communicated to all the other agents. In a scenario where some of these deductions are computed quicker than others, it can lead to significant speed up for the other agents which can now use these deductions within their processes to reduce their problem and hence to prune their search space.

As mentioned before, the architecture employed in the DPLL-Stålmarck solver provides a useful abstraction of how external information can be utilised by the DPLL solver. Within our framework, the Stålmarck agent can be replaced by any agent that can provide some information that is universally relevant to all parts of the search space of the DPLL problem. A dedicated heuristic supplying agent can be plugged in with minimal work and can readily communicate with the other agents. This will enable the introduction of more reactive approaches as agents from different parts of the search space can communicate with each other. As illustrated by the DPLL-Stålmarck implementation, it is possible to use multiple solvers from within the program without having to resort to an operating system level communication.

8.5. Resource management, Fault-tolerance, Start-stop facilities

The stålmarck agents in our work are independent of the DPLL solver in terms of data-state and the stage of execution and the deductions made by stålmarck are applicable across the board to all parts of the DPLL solver as well as to other stålmarck agents. Thus, the stålmarck agents can be orchestrated to make the most of idle resources and
since they do not affect the correctness or completeness of the DPLL, the orchestration can be made in an extremely flexible manner. In our current implementation, there is an option to vary the number of Stålmarck agents which can be used to effectively implement load-balancing strategies. Also, the failure of one of these threads does not compromise the soundness of the algorithm.

As discussed in [14], start-stop-resume facilities are important for tackling large/difficult instances [14] and are particularly relevant for GRID-like environments. The Stålmarck agents can prove to be good candidates for such a setup as their deductions are independent of the state of the search space.

8.6. Abstractions

Of particular interest to this project are possible abstractions and concurrent design patterns that emerge, which are generic enough to be extended to other scenarios and other solvers, as individual abstractions or as compositions of abstractions. Some ideas on relating the techniques employed in the current work to possible extensions to other forms of theorem proving are as follows:

- Inbox as an abstraction to hold information and which can be subscribed to information-providing services. The dpllInbox is an instance of this abstraction.

- OR parallelism, with facility for killing one thread when an answer has been found by another thread; The fastest-first function used in the distributed-DPLL is an instance of this abstraction.

- Producer-consumer behaviour: The Stålmarck agents are illustrative of this aspect

- Collaborative/co-operating agents: for example, the Stålmarck agents are helping the DPLL solver.

- Message passing mechanisms. The Stålmarck agents post their results to D-s as well as to its subscribers. In the DPLL-Stålmarck implementation, the DPLL agent is a subscriber to the Stålmarck agent service and it gets notified of the results. This is achieved via message-passing.

9. RELATED WORK

Recently, there has been a lot of attention within the SAT community in applying parallelisation techniques to SAT solving. In this section, we present an overview of some of these approaches and discuss the context of our work in relation to them. The design and techniques of these approaches can be grouped based on the target hardware environment and the core communication mechanisms provided by the platform.

PaSAT [13], ySAT [6] are systems that have been primarily targeted at Multi-core and multi-CPU architectures. They have tightly integrated SAT Solving units and are thread-based implementations. They have a shared memory for communication and clause database which in turn provides high bandwidth, low latency and low overhead communication. DPLL based solver is the core engine with communication of clauses learned by conflict-driven clause learning mechanisms.

PSATO [14], GridSAT [4], zetaSAT [3], PaMira [11] have been specifically designed for a network of workstations. All these systems are based on the DPLL based SAT solvers. They are highly scalable and can be used across 100s of workstations in parallel. They implement loose integration of SAT solving units. They adopt a master-client model of distribution. Each client has its own clause database. They rely on message passing for communication. This imposes a further restriction on the choice of messages that get communicated. For example, in some of these systems, only clauses with few literals are shared as it means less burden on the message passing facility. Further policies have to be used on the client side on which clauses it wants to share.

The work presented in this paper differs from these efforts in the following ways:

- Not focussed primarily on DPLL
- Application of concurrency/distribution to a Stålmarck based solver
- Application of concurrency/distribution to engineer a hybrid solver, which as far as we know has not been attempted so far
- The emphasis is on inter-process communication and collaborative problem solving rather than on raw parallelisation
- The lightweight framework that allows for rapid-prototyping and ease of experimentation
- Use of a functional programming language for implementation

10. CONCLUSIONS

In this paper, we have discussed our experiments with applying concurrent/distributed techniques to SAT, using
a concurrent/distributed functional programming language called Alice. We have applied these techniques to existing sequential algorithms (DPLL, Stålmarck) and engineered a hybrid solver (DPLL-Stålmarck). We are working on a thorough evaluation involving metrics of time, search-space-size and the number-of-stålmarck-agents.

An important contribution of this project is the hybrid solver. This architecture would not have been feasible in a sequential setup. Furthermore, the architecture can be extended to other information-providing agents as well, which is clearly a new direction to pursue for future SAT solvers. This is also a good place to mention that this research direction can open up fresh new approaches that were hitherto not feasible. Application of parallelism, and concurrent/distributed programming techniques is applicable at different levels as illustrated by our experiments, ranging from fine-level granularity at choice-points to a coarse-level of granularity in the hybrid solver.

We intend to work on optimising the solvers for multicore architectures and conducting further empirical evaluation on the solvers developed. Preliminary test results have showed an interesting artefact of a relation between the number of stålmarck agents and the assignments produced for SAT cases. We hope to investigate this phenomenon more closely. We also feel that there is a lot of scope for these asynchronous and collaborative problem solving techniques to be employed in incomplete SAT solvers, adopting probabilistic approaches.

It is worthwhile pointing out that the work described in this paper is part of an ongoing project investigating the efficacy of use of concurrent/distributed programming techniques to theorem proving. As part of this effort, we are also working on applying these techniques to tactic-based first order theorem proving.

REFERENCES


