CS 690LG Project: Distributed SAT Solving

Prateek Sharma

December 18, 2015

1 Introduction

The Boolean satisfiability problem (SAT) has many real-world applications and solving it is extremely important. Although it is NP complete, even large SAT instances are now solvable, mainly due to advances in heuristics and a better understanding of the structure of SAT problems. SAT has a growing list of applications. Traditionally, it has been used for hardware and software verification (like model checking) and general theorem proving (prover9/mace4). More recently, SAT solvers are being used in areas ranging from computational biology to bitcoin mining.

Since SAT is NP-complete, the only way to solve it is a worst-case exponential search. Let an SAT instance be denoted as (V,C), where V is the set of variables $\{x_1,x_2,\ldots x_n\}$, and C is the set of clauses. One of the original SAT solving approaches was DPLL [**?**], which is a basic backtracking search in this exponential search space. The DPLL approach has proven to be remarkably useful, and even after more than 50 years, some industrial SAT solvers still use some variant of it, although Conflict Driven Clause Learning (CDCL) is increasingly popular.

The large computation time required to solve SAT instances has been conveniently provided by Moore's law and Dennard Scaling, which resulted in faster clockspeed. However, recent hardware architectures have strongly veered towards more parallelism in the CPUs (via multiple CPU cores). To make use of multiple CPU cores, recent SAT solvers such as plingeling [4] and ManySAT [6] have incorporated multi-threading into traditional solvers. For example, ManySAT is a multi-thread version of MiniSAT, a well-known single-threaded SAT solver. Multithreaded sat solving is fairly well studied [9].[7] has a good discussion on the challenges in parallel SAT solving.

There are two main challenges in parallel SAT solving: problem decomposition, and communication. In order to address these challenges, there are two main techniques used by parallel SAT solvers:

1. **Portfolio Approach:** Each solver thread operates on the entire input problem, but with different heuristics. Threads therefore "compete" with one another to solve the same problem instance. The portfolio approach at first seems wasteful and naive, because it attempts to explore a large search space by running the multiple searches in parallel and hoping that one of them gets lucky.
2. **Divide and Conquer:** Either the problem instance or the solution space is divided among the threads, and each thread thus operates on a different search space. Division of the search space is non trivial, and balancing the workloads of multiple threads is a nuisance in this approach because some threads can be stuck with a "hard" portion of the search space. Partitioning techniques are explored in [8].

Throwing more computing resources at the problem seems to be a viable strategy to solve SAT problems, and using all the CPU cores is certainly a step in that direction. We can extend this approach, and opt to use a cluster of machines instead of a solitary machine. Clusters of commodity hardware is the most popular and cost-effective option to scale the computing resources. The ubiquity and low-cost of cloud computing services such as Amazon's EC2, where users can launch a cluster of 1000s of machines in a minute at a very low cost, also means that the cloud is a cost-effective platform for computationally intensive tasks like SAT solving.

Thus, cloud platforms which offer clusters on-demand at low costs can serve as an ideal platform for distributed SAT solvers. However, there are a number of challenges in implementing a fast, distributed solver. The primary

difference between a parallel SAT solver and a distributed solver is that a parallel solver can take advantage of shared memory within the machine. That is, all threads have low-latency access to shared data-structures such as shared clauses. In a cluster, shared state is minimal and slow, and we have to rely on some form of message passing. Even if techniques like distributed shared memory are used, the networking latency and other synchronization overheads make sharing data a much more expensive operation when compared to the single machine case. It must be noted that these concerns **also** apply to single machine parallel solvers with NUMA architectures, cache-line false sharing, cpu cache contention among threads, etc. But they are exacerbated in a distributed environment.

Distributed SAT solvers have been around for some time and most of them rely on message passing. Examples of such solvers are gridsat [5], pasat, and paMIRA [10]. Some of the performance bottlenecks and optimizations for parallel solvers also apply to distributed solvers.

1.1 Project Scope & Contributions

In this project report, I will study and extend ManySAT [6], a parallel SAT solver. ManySAT is a DPLL portfolio based solver which utilizes multiple threads. I will show the performance characteristics of ManySAT on SAT benchmarks, and compare it to a recent world-champion solver, plingeling [4].

ManySAT is restricted to working on a single machine, and is not capable of distributed operation. I will also develop a distributed version of ManySAT, called **dManySAT**, which can utilize a cluster of machines. I will show the design and implementation of my distributed version, along with preliminary performance results.

2 Background: ManySAT

ManySAT is a DPLL (backtracking) based solver which spawns multiple threads, all attacking the same search space. Each thread, however, has different heuristics, which enable the search to proceed in different order. Below is a description of the various components of the solver:

Input: ManySAT takes a SAT instance in the DIMACS format as the input, along with other solver configuration parameters. The input instance is parsed and fed to SatElite for pre-processing.

Parallelism: Parallelism is provided by using multiple *threads* to solve the input instance. The number of threads can be user specified, but defaults to the number of cores available on the machine. Threads are implemented by using the OpenMP API. Each thread is given the entire input problem instance to solve, and the program terminates if any thread has finished solving the instance (either a satisfying assignment or proving that it is Unsatisfiable).

Portfolio: ManySAT originally employed a portfolio of heuristics for the different threads. For example, different threads have different restart and decision variable choosing heuristics (VSIDS). However, the recent ManySAT implementation (version 2.0), also provides a *deterministic mode* of operation, in which all the threads use the same heuristics, except for the *first* decision variables. That is, the threads use different decision variables at the first step of the search, and then all use the same heuristic with the same parameters and restart policies. This deterministic mode enables better performance comparisons, but at the cost of portfolio diversity, which is also crucial for massively parallel SAT solvers. However, ManySAT is targetted to parallel solving on commodity machines with 4-8 CPU cores. My study of its source code leads me to believe that there is a lot of scope for improvement, especially by being NUMA and cache-line false-sharing aware.

Restart: If a thread is "too deep" into the search space, then the solver restarts its search (from the very beginning). Restarts are a crucial component of all modern SAT solvers, and avoid the solvers from getting stuck in some useless part of the search space. ManySAT has implemented many restart heuristics, and uses the Luby [**?**] policy by default.

Clause learning: During the course of a search, ManySAT uses a CDCL-like scheme for learning clauses. These learnt clauses are used when the thread is restarted.

Clause sharing: Knowledge sharing is the crucial component of parallel solvers, and allows threads to learn from each other. Clause sharing in ManySAT is performed just before a thread restarts. The set of learnt clauses is kept in a global (shared by all threads) structure. Before a restart, a thread exports the clauses learnt in the prior run and imports

clauses from other threads, applies these learnt clauses to its local input instance, and then restarts the search, armed with new and potentially useful knowledge.

The flip-side of sharing knowledge is that the number of sharing the clauses can be detrimental to performance because of the overheads involved with sharing and storing all the shared clauses. Pruning the shared clauses is essential to reduce the overhead due to sharing and memory requirements. ManySAT addresses this by providing user-controllable configuration parameters to limit the number of clauses imported by threads from other threads. If the limit is crossed, then no more clauses are shared. This limit is applied pairwise among threads, so a thread may not accept any new clauses from thread-A, but can do so from thread-B, if the number of shared clauses from thread-B is still under the limit.

Another mechanism provided by ManySAT to control clause sharing is to restrict the size of the clauses shared. The user can specify (via the `limitEx` option, the maximum size of the clauses considered for sharing. The default maximum size of shared clauses is 10. The relation between performance and clause sharing will be evaluated in the evaluation section. Clause sharing is an important design component and will be explored in depth later when discussing the distributed ManySAT implementation.

3 Design of dManySAT

This section describes the distributed solver developed as part of this course project. The solver is based on ManySAT, and gives it the ability to operate in a distributed manner (hence the name). Since ManySAT is portfolio based, it is easier to parallelize. The threads operate independently for the most part, only communicating before restarts when they exchange clauses. The rest of this section describes the clause sharing.

Interface: The command-line based user-interface of ManySAT remains. dManySAT is launched on all the nodes of the cluster with the same problem instance.

3.1 Communication

ManySAT threads do not communicate much with each other, which is good news for the distributed implementation.

Thread Start: Thread starts across different machines are not synchronized. Each solver starts independently, there is no barrier.

Finish: When a thread has found the answer, it terminates the solver on the local machine. The result is copied to the other machines, and their solvers are terminated via `ssh`.

3.2 Clause Sharing

Clause sharing among different threads on the same machine is the same as ManySAT. Unit clauses and clauses of longer length are shared among threads before restarts via a shared in-memory data structure. In ManySAT, this is a 2-d array of pointers to vectors of clauses. For each pair of threads (s,t), there is a vector of clauses that s has learnt from t. The clause sharing among local threads has not been modified. The shared data structure is protected by an OpenMP barrier.

Clause sharing between machines is the interesting part, and received a significant portion of the effort in the design and implementation phase of this project.

The basic design principle for clause sharing is fairly simple. On each machine, one thread from the thread-pool is "stolen" from the solver, and instead acts as the communication thread. This communication thread sends and receives clauses across the network. When the threads on a machine restart, they perform their local clause sharing as usual. After the local clause sharing is over, they then perform clause sharing with other machines (via the communication thread) as follows.

Exporting clauses to other machines is performed by broadcasting the clause to all the machines. On each machine, the communication thread receives and stores these broadcast clauses.

Once learnt clauses have been exported, a thread then *imports* clauses from other threads. Here too, clauses from local threads are imported first. Then, the clauses are imported from the communication thread. Remember that the communication thread receives broadcasts from all the threads from all the machines. During the import, the local threads pull the clauses from the communication thread.

The communication thread maintains a buffer of received clauses which is cleared when local threads have already merged these clauses into their instances. In ManySAT, all clauses are shared between all threads (as long as they are under the sharing quotas). However, this threatens to overwhelm threads with too many learnt clauses, because the communication thread has a large number of clauses from all the machines in its buffers.

This "too many shared clauses" problem is mitigated by using two techniques:

1. The input buffer in the communication thread is of a fixed size and maintained in a FIFO manner.
2. Instead of sharing the clauses with all the local threads, dManySAT only shares the remotely-learnt clauses with k of n threads. The first k solver threads to ask the communication thread for clauses are given all the clauses, and then the input buffer is cleared. This has two advantages. First, not all threads are burdened with a large number of learnt clauses. Second, the buffer is more frequently cleared because we dont have to wait for all the n threads to ask for the shared clauses. In the current implementation, $k = n/2$.

4 Implementation

dManySAT is implemented by modifying ManySAT, and requires about 500 lines of C++ code. The code is available on my github repository [1].

Communication among machines is done over the network by the communication threads. Many communication platforms were contemplated, tried, and abandoned: MPI, zeroMQ. MPI is the defacto HPC communication interface, but sadly proved to be too restrictive when combined with ManySAT's existing clause sharing design. The primary goal of the implementation was to minimize the amount of changes to ManySAT. The solution finally adopted was to use simple BSD TCP sockets to send and receive clauses. The communication thread maintains a list of open sockets to the other machines (the peers), and broadcasts clauses by sending them to all the peers.

Since clauses are sent over the network, care it taken to serialize them and convert from host to network byte order, etc. ManySAT clauses are simply a vector of literals. ManySAT has its own vector implementation (not `std::vector`), which is quite amenable to network communication and did not require a serialization library like protobufs. A manysat vector is simply an array with the size as the first element. Thus, to send a clause, the clause vector is just sent over the wire. Unit clauses are represented by literals, which are again plain integers. ManySAT uses a single integer to encode both the variable and the polarity. Unit literals are converted to unit clauses (by prefixing the size=1) and then sent.

The communication thread constantly listens on its receive socket, and places all clauses on the input buffer. There are separate sockets for receiving and broadcasting. The input buffer is filled in by the communication thread, and drained by the other solver threads, and is protected by an OpenMP mutex.

Exporting a clause implies a message-send, which is simply a socket send after converting to a safe binary format. Currently, dManySAT broadcasts clauses eagerly without any buffering, which adds to the networking overhead. A simple optimization would be to buffer exported clauses and send bigger messages.

5 Evaluation

To evaluate the performance of ManySAT, I used the Sat-Competition 2014 [3] benchmarks. In particular, since SAT solvers are usually used for specific applications, I used the "application" benchmarks from the benchmark suite [2].

A major problem with SAT is that many instances require too much time. When I ran the full application benchmark suite to sanity-check ManySAT, some problems ran for over two days(!). For the rest of the experiments, I have used a timeout value of 1000 seconds, and problems taking longer than that are terminated. From the full benchmark

suite, only **26** problems finished within 100 seconds, and all the experiments were performed on each of these 26 problems. The summary statistics of these 26 problems is shown in Table 1.

	Average	Maximum
Variables	11,000	53,000
Clauses	60,000	161,000

Table 1: Number of variables and clauses for the 26 input problem instances

The hardware platform used for all the experiments was a Dell PowerEdge R210 server with 16GB memory and 8 core Xeon E3-1240 V2 CPU. This CPU has hyperthreading enabled (thus 8 effective CPUs). Some of the experiments were also performed on a newer, 6th generation Intel CPU (Core i5-6500) to test whether ManySAT can take advantages of the newer architecture and new instruction set. The ManySAT program was compiled by GCC with the `-O3` optimization flag.

plingeling: In addition to evaluating my modifications to ManySAT, I also compared with plingeling, the parallel version of lingeling, which is the 2011 SAT contest winner and a very competitive industrial SAT solver. Plingeling is CDCL based, and only shares unit clauses among threads.

5.1 Clause sharing's effect on performance

To see the relatation between clause sharing and performance, ManySAT was run on the 26 SAT instances and run for a maximum duration of 1000 seconds. The results are shown in Figure 1. A better representation is the CDF of running times, which is shown in Figure 2. The boxplot of the running times is shown in Figure 3. Finally, Figure 4 shows the relation between the percentage of clauses shared (relative to number of clauses in the input) to the running time for one particular input problem. There doesn't appear to be a clear trend for this particular problem instance, but sharing clauses of length upto 10 seems to yield the best results overall 3.

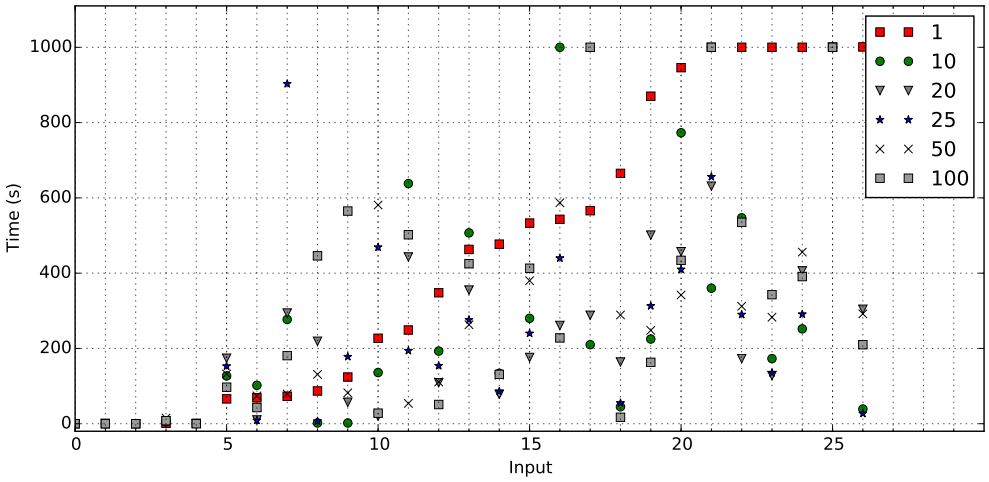

Figure 1: Scatter-plot of running times for ManySAT with different clause sharing parameters. The labels indicate the max length of clauses shared between threads.

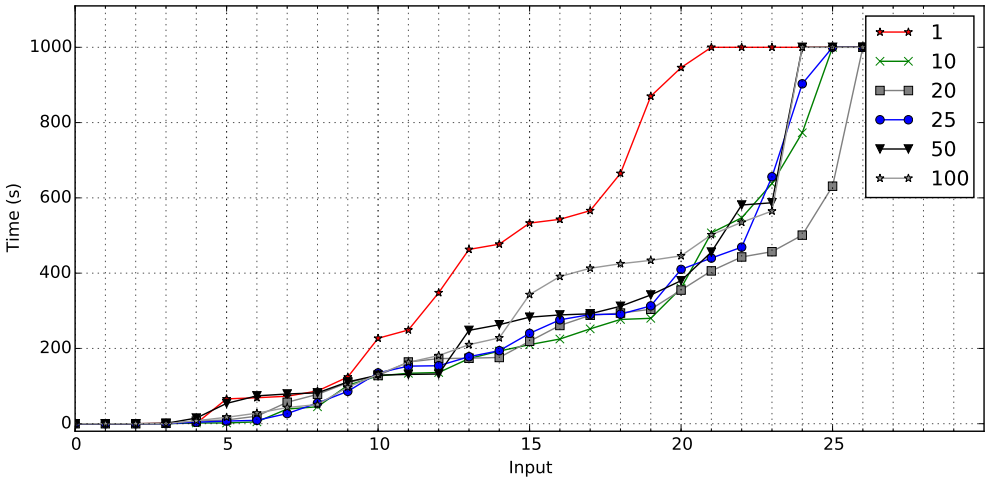

Figure 2: CDF of running times of ManySAT with different clause sharing limits

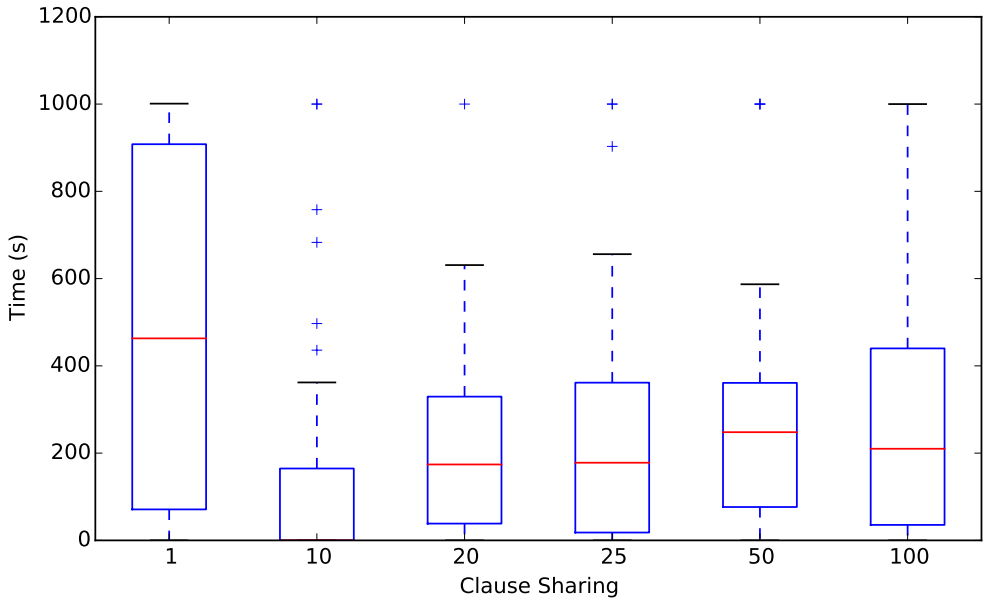

Figure 3: Summary of running times. Default clause length of 10 yields best results. Sharing only unit clauses hurts performance significantly.

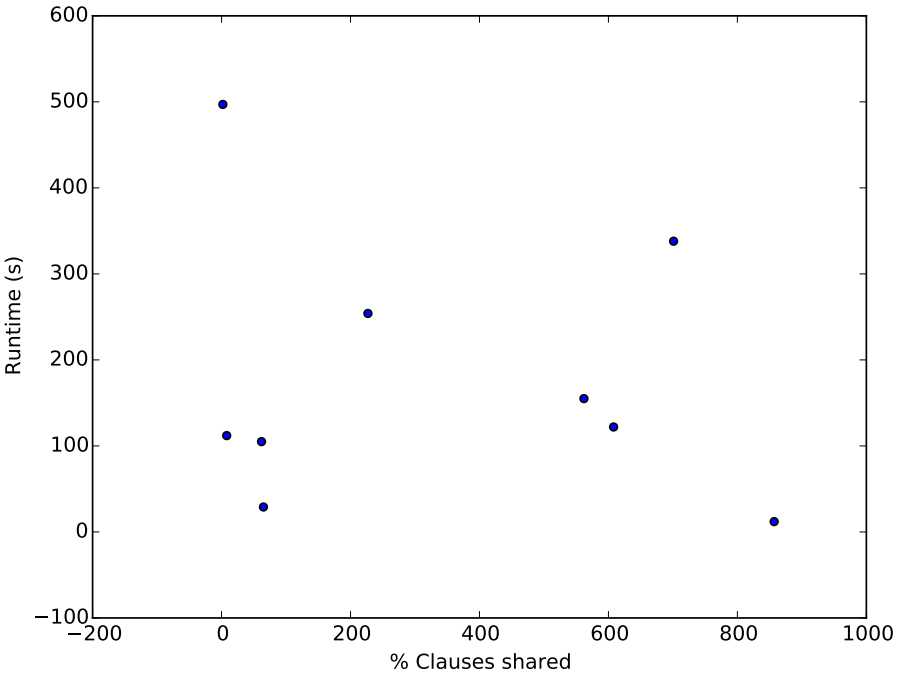

Figure 4: Percentage of clauses shared vs running time for gss-18-s100 input

5.2 Distributed ManySAT performance

To evaluate distributed ManySAT, it was compared to vanilla ManySAT (can only run on a single machine), and plingeling (also on a single machine). For dManySAT, two nodes were used with 4 cores each. To reduce the effect of network latency, both the nodes were on the same physical machine. ManySAT and plingeling used 8 threads, and dManySAT used 2 instances with 4 cores each. Thus, the compute resources used by all configurations are the same, and in the case of dManySAT, the nodes have to use message passing over sockets instead of shared memory.

The performance is summarized in Figure 5. The distributed version of ManySAT is noticeably slower than vanilla ManySAT, and plingeling is the fastest of them all, which makes sense because it is afterall the world record holder in the SAT competition. This experiment was conducted to capture the overhead of TCP-based message passing, and the overhead is only moderate. Note that distributed ManySAT can increase the computing power available to it by using clusters of machines, something which the other local solvers cannot do.

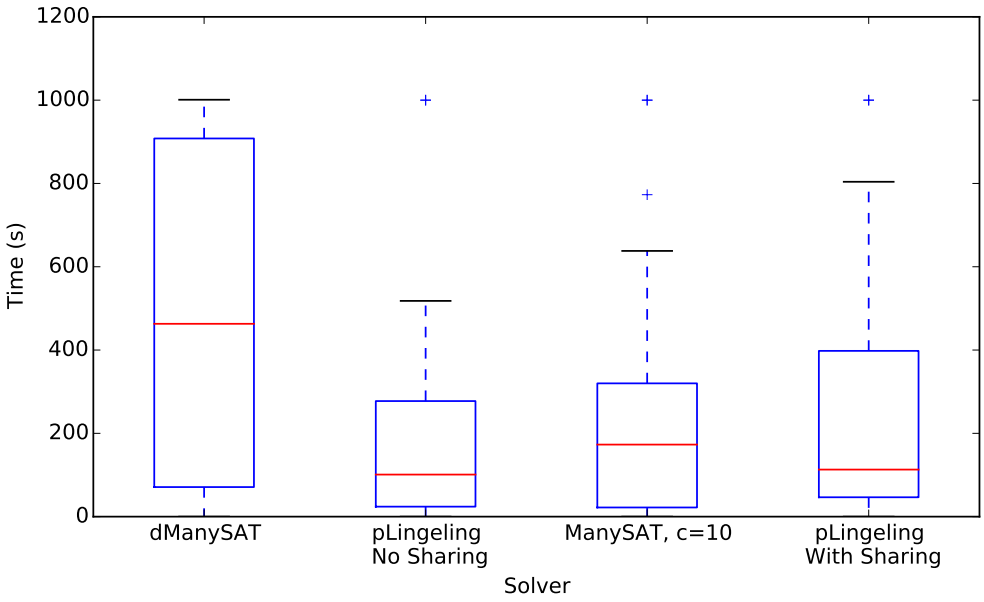

Figure 5: Distributed ManySAT, ManySAT and plinegling performance. Distributed ManySAT was run with two instances of 4 threads each. ManySAT and plinegling used 8 threads.

6 Conclusions & Future Work

In this project, I showed the feasibility of converting a parallel solver into a distributed one. Clause sharing is extremely important for performance, but too much of it is a bad thing.

The distributed-ManySAT developed uses message passing between nodes to share learnt clauses. The performance of dManySAT is surprisingly good, especially given that the implementation is missing some very obvious optimizations. As part of future work, buffering learnt clauses and batching them to increase message sizes, is the most important and crucial step. Replacing standard UNIX sockets by other direct hardware access mechanisms (like NetMap) is also a potential mechanism to reduce latency and message processing overheads.

A more thorough evaluation of the performance would have been nice, but the entire test-suite took about 5 hours to run, which made running experiments a time consuming process.

References

[1] dManySAT github repository. `http://github.com/prateek-s/manysat2.0/`.

[2] SAT Competition 2014 Application Benchmarks . `http://www.satcompetition.org/2014/files/sc14-app.tar`.

[3] SAT Competition 2014 Benchmarks . `http://www.satcompetition.org/2014/downloads.shtml`.

[4] Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[5] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37. ACM, 2003.

[6] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[7] Youssef Hamadi and Christoph Wintersteiger. Seven challenges in parallel sat solving. *AI Magazine*, 34(2):99, 2013.

[8] Antti EJ Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 372–386. Springer, 2010.

[9] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pages 926–931. IEEE, 2007.

[10] Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamira-a parallel sat solver with knowledge sharing. In *Microprocessor Test and Verification, 2005. MTV'05. Sixth International Workshop on*, pages 29–36. IEEE, 2005.