Parallel WalkSAT with Clause Learning

Austin McDonald
Machine Learning Department
Carnegie-Mellon University
Pittsburgh, PA 15213
austinm@cs.cmu.edu

Advisor: Geoff Gordon
Machine Learning Department
Carnegie-Mellon University
Pittsburgh, PA 15213
ggordon@cs.cmu.edu

Abstract

We present an extension of WalkSAT, a stochastic local search algorithm for solving SAT problems. Our extension learns new clauses by resolving existing clauses based on the current state of a WalkSAT run. We show that clause learning leads to significant speedup in WalkSAT runs, both in terms of fewer steps and faster runtime. We argue that our WalkSAT implementation is easily parallelizable, and we demonstrate great speedup from parallelization by leveraging learned clauses across the entire set of parallel runs. WalkSAT, due to its simple algorithm, is particularly well-suited to implementation on graphics processing units (GPUs), which are often capable of many more operations per second than a CPU for applications with little control flow. We describe the modifications necessary to extend our algorithm to work in a GPU environment.

1 Introduction

Satisfiability problems are an important field in computer science. The basic satisfiability problem (SAT) is: given a Boolean expression, is there some assignment to the variables that makes the expression true? SAT is the defining problem in the complexity class of NP-complete problems; all NP-complete problems can be reduced in polynomial time to 3-SAT, and vice-versa. Specifically, general SAT problems can also be reduced to 3-SAT. In addition, many important problems across computer science can be expressed as SAT problems, such as planning and probabilistic planning problems [8], relational learning [3], and computer-aided processor design [13].

In particular, a SAT solver is a strong step towards a MaxSAT solver [2] where clauses are given weights, and we seek a solution that maximizes the sum of the weights of the satisfied clauses. MaxSAT is an important part of inference in many important problems, such as probabilistic planning problems and Markov Logic Networks [9].

Algorithms for solving SAT problems are also useful for solving problems in the class #P. According to Stockmeyer [11], any problem in #P can be approximated within a polynomial factor in bounded probabilistic polynomial time, given an NP oracle. Problems in #P include exact inference in Bayesian networks and Markov random fields.

WalkSAT [10] is a stochastic local search algorithm that tries to find satisfying solutions to Boolean expressions. In this paper, we extend WalkSAT with an algorithm that learns new clauses that are relevant to the area of the search space WalkSAT is currently exploring. We then explore the effects of performing multiple WalkSAT runs in parallel and leveraging learned clauses across all runs.

Extending algorithms to run on parallel architectures is increasingly important as chip manufacturers move from increasing clock speeds to increasing the number of cores on a processor. Clock speeds have remained relatively steady on processors for the last few years, as chip manufacturers reach the point where physical constraints cause multiple cores to be a more efficient investment of power

than increasing clock speeds. In order for algorithms to continue to improve, they need to exploit this trend by scaling well across multiple processors.

WalkSAT is particularly well suited to parallelization; it can be trivially parallelized by simply running multiple copies, which results in some speedup by taking the fastest of n runs. However, we would like some method for sharing computation between parallel WalkSAT runs. Our solution entails learning clauses in each run and sharing the learned clauses between runs. We show that learning clauses not only speeds up individual runs, but results in further gains when leveraged across parallel runs.

Since the WalkSAT requires only minimal control flow, we believe it may be well suited to parallelization on simplified processors like GPUs; we present an analysis of the algorithm that suggests this. Clause learning is common in other SAT solvers, but our application to WalkSAT is novel. The extension to WalkSAT is important since other SAT solvers do not appear to lend themselves as easily to running on GPUs.

2 SAT

Given a Boolean formula of variables joined by AND, OR, and NOT connectives (and parentheses), we wish to find a solution (if the formula has one). The canonical representation for Boolean expressions is conjuctive normal form (CNF), where the formula becomes a conjuction of disjunctions of literals (a variable is any symbol in the formula, such as A or B below; a literal is a variable or its negation). Any Boolean expression can be expressed in CNF through a sequence of reductions. For example by distributing $\wedge$ over $\vee$,

$$(\neg A \wedge B) \vee (C \vee D) = (\neg A \vee C \vee D) \wedge (B \vee C \vee D)$$

This form is particularly convenient because, to judge the truth of the entire expression, we need only ensure that each term in the conjunction (known as a clause) has at least one literal true. Naive algorithms for performing this reduction result in exponential increase in the number of clauses; however, Tseitin [12] showed the existence of a transformation that preserves satisfiability, though not equivalence, that only increases the number of clauses and variables linearly. The transformed clauses will be satisfiable iff the original clauses were satisfiable.

Any propositional logic formula can be reduced to CNF, and under some assumptions (e.g., a finite, known universe, unique names, and known functions) first-order logic can be reduced to CNF.

2.1 Resolution

Given a conjuction of clauses, we can often perform some additional operations that produce new clauses while maintaining the meaning of the formula. Of particular interest is resolution, which allows us to combine two clauses to produce a third clause. The resolution rule is simple:

$$(a_1 \vee ... \vee t \vee ... \vee a_n) \wedge (b_1 \vee ... \vee \neg t \vee ... \vee b_n) \Rightarrow (a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_n)$$

This rule lets us take any two clauses, where one contains a literal t positively and the other contains t negatively, and combine them to produce a third clause which does not contain t but contains all of the other original literals.

Many techniques for solving SAT problems rely on resolution. For example, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [5], a backtracking SAT solver, uses resolution on clauses of unit length (called "unit resolution") to propagate constraints. And, many DPLL-style solvers use conflict-driven resolution to learn new, useful clauses. We hope to generalize this method here.

3 WalkSAT

In contrast to the DPLL family of backtracking solvers, WalkSAT [10] is a stochastic local search. It operates by maintaining an (initially random) assignment to all the variables. At each step, it chooses

an unsatisfied clause. With probability p, it flips a random variable from the chosen clause. With probability $1-p$, it flips the variable of the chosen clause that maximizes the decrease in the number of unsatisfied clauses (which could be negative). The algorithm is given in detail in Algorithm 1.

Algorithm 1 Detailed WalkSAT algorithm

```
function WalkSAT(assignment : Bool[n], clauses : Int[c][n], p_random_step : Real)
num_unsatisfied_clauses ← countUnsatisfiedClauses(assignment, clauses)
while num_unsatisfied_clauses > 0 do
  unsatisfied_clause ← getRandUnsatClause(clauses, assignment, num_unsatisfied_clauses)
  uniform_sample ← randomReal([0, 1])
  if uniform_sample < p_random_step then
    random_index ← randomInteger(1, len(unsatisfied_clause))
    var_to_flip ← abs(unsatisfied_clause[random_index])
  else
    var_to_flip ← getBestVarToFlip(assignment, clauses, unsatisfied_clause)
  end if
  assignment[var_to_flip] ← ¬ assignment[var_to_flip]
  num_unsatisfied_clauses ← countUnsatisfiedClauses(assignment, clauses)
end while
return assignment
end function
```

This function takes a starting random state, the clause database, and the probability of a random step, and produces a satisfying assignment. It can easily be augmented to return after a set number of steps or other termination criteria. An *assignment* is an array of n Boolean variables, and a clause is an array of n Integers. The *clauses* database is an array of c clauses. In a clause of length k, the first k values are nonzero, and the remainder are 0. To represent a clause, we store the indices of the variables in that clause, and store them negatively to represent negation (we assume arrays are 1-indexed).

Algorithm 1 does not specify implementations for basic functions such as countUnsatisfiedClauses. Here we present simple, rather than optimized, versions of these functions. countUnsatisfiedClauses simply iterates through the clause list, counting the number of unsatisfied clauses. getRandUnsatClause chooses a random unsatisfied clause from the clause database; it can be implemented by choosing $n \in (0, ..., \textit{num\_unsatisfied\_clauses})$ and scanning the clauses until the nth unsatisfied clause is found. getBestVarToFlip is the most complicated of these functions. It requires us to scan through the unsatisfied clause, and, for each literal in that clause, scan through the list of all clauses and count the change in the number of clauses that would be satisfied by flipping that variable. This is simply two nested loops.

Some particular optimizations include replacing numUnsatisfiedClauses by a counter and updating it appropriately at the end of the loop; passing this counter to getRandUnsatClause to avoid recalculating it; and building a map of variables $\mapsto$ clauses, so that in getBestVarToFlip we can simply look up the clauses a variable appears in when we decide whether to flip it, instead of repeatedly scanning all the clauses for each variable. Further improvements can be gained by use of the "watch literal" trick, where we choose a positive literal from each satisfied clause, and only recalculate the clause's status if we flip that variable (since it can't go from satisfied to unsatisfied without flipping that literal). Finally, we can change getRandUnsatClause from a scan through the entire clause database to an index into a vector of unsatisfied clauses. However, we must then keep that vector updated at the end of the loop. Many more possible optimizations exist in the literature [14].

4 Clause Learning

Many popular SAT-solving algorithms, especially those based on DPLL, learn new clauses as part of their operation. We would like to extend this functionality to WalkSAT. Conflict-driven clause learning algorithms, such as the ones found in Zhang et. al. [14] and Beanie et. al. [1], maintain a set of assumptions, as well as keeping track of which clauses forced which assignments. Since

conflict-driven algorithms maintain only a partial assignment, they can stop as soon as they make an assumption which causes a conflict (a contradiction) and backtrack.

4.1 Conflict-Driven Clause Learning

When a search algorithm reaches a conflict, it is often useful to learn a clause covering this state to prevent reaching it again later. We call variables chosen by assumption *decision* variables and those chosen by unit propagation *implied* variables, with decision and implied literals defined similarly. Here, "assumption" refers to the branching choices made by DPLL, where DPLL arbitrarily sets a variable to True or False and propagates the results; these are the steps that it can backtrack on. "Unit propagation" is the process of iteratively using unit resolution to propagate constraints, also a core part of the DPLL algorithm. To learn a new clause, we first build an *implication graph*. The implication graph will represent all of the reasoning possible in the current state, including the chains of inference that led to any contradictions we may have. Initially, we seed the implication graph with the decision literals as nodes. These nodes will have no edges pointing to them. Next, we iteratively look for clauses of the form $(a_1 \vee ...a_k \vee a)$ where $\neg a_1 ... \neg a_k$ already appear in our implication graph. We then add a to the graph, with an edge from each literal in the clause to the new node, and label these edges with the clause. We also add a conflict node Λ; any variable which causes a contradiction (appears both positively and negatively) gets an edge to this node (from both occurrences). Note that we can incrementally build this graph during our search, or by looking at our search tree.

To learn a clause from an implication graph, we choose a subgraph known as the *conflict graph* (see Figure 1). The conflict graph is a subgraph of the implication graph containing just the inference chains that led to a single contradiction. At any point, we may have multiple contradictions; however, for clause learning, we need to focus on just one. The conflict graph contains the conflict node Λ, exactly one conflict variable (one node for each sign), and requires that all nodes in the conflict graph have a directed path to Λ. Any cut in this graph which separates the decision variables from the conflict node and at least one conflict variable (of either sign) will result in a new clause formed by negating all literals who have an outgoing edge crossing the cut. The choice of the conflict subgraph and which cut to take is left up to the implementer, and is the cause of much debate. The learned clause is consistent with the other clauses, due to the fact that it can be derived via resolution from the set of clauses [1].

4.2 WalkSAT Clause Learning

We wish to adapt this clause learning algorithm to a WalkSAT setting. We generalize the idea of a conflict graph as follows: choose a random variable from an unsatisfied clause; add both its positive and negative instances to the conflict graph with edges to Λ, and search for clauses implying those two variables. We map the concept of *decision* variables to those chosen by random steps, and *implied* variables to those chosen by the maximization steps. We also choose to learn a clause based on the clause randomly chosen by WalkSAT at a given step (a random unsatisfied clause) which we call the *conflict clause*.

We do not explicitly build an implication graph; instead, we implicitly maintain a subset of the implication graph by storing the clause that forced the current setting of a variable (or nothing, if it was randomly flipped). We arbitrarily choose one literal x from the conflict clause and add it to the graph, with an edge to Λ. Combined, the remaining literals must imply $\neg x$; we add a node for $\neg x$ with an edge pointing to Λ, and nodes for the remaining literals, with edges to $\neg x$. Then, for each of these nodes with no parents, we choose some satisfied clause that contains this leaf positively. In our implementation, we choose the specific clause that was chosen at the maximization step that forced that particular variable; however, any satisfied clause will do. We add as parents all of the literals in this satisfied clause and label the edges collectively with the clause. We continue expanding nodes in this fashion until an arbitrary termination criterion is satisfied. In our implementation, we terminate when we we reach decision nodes. We also terminate if we reach a variable that was chosen by a maximization step, if it is no longer implied by the clause that originally caused us to set its current sign.

This gives us a similar chain of inference to the conflict-driven clause learning algorithm. To learn a clause from our conflict graph, we take any cut which separates Λ and at least one of the conflict

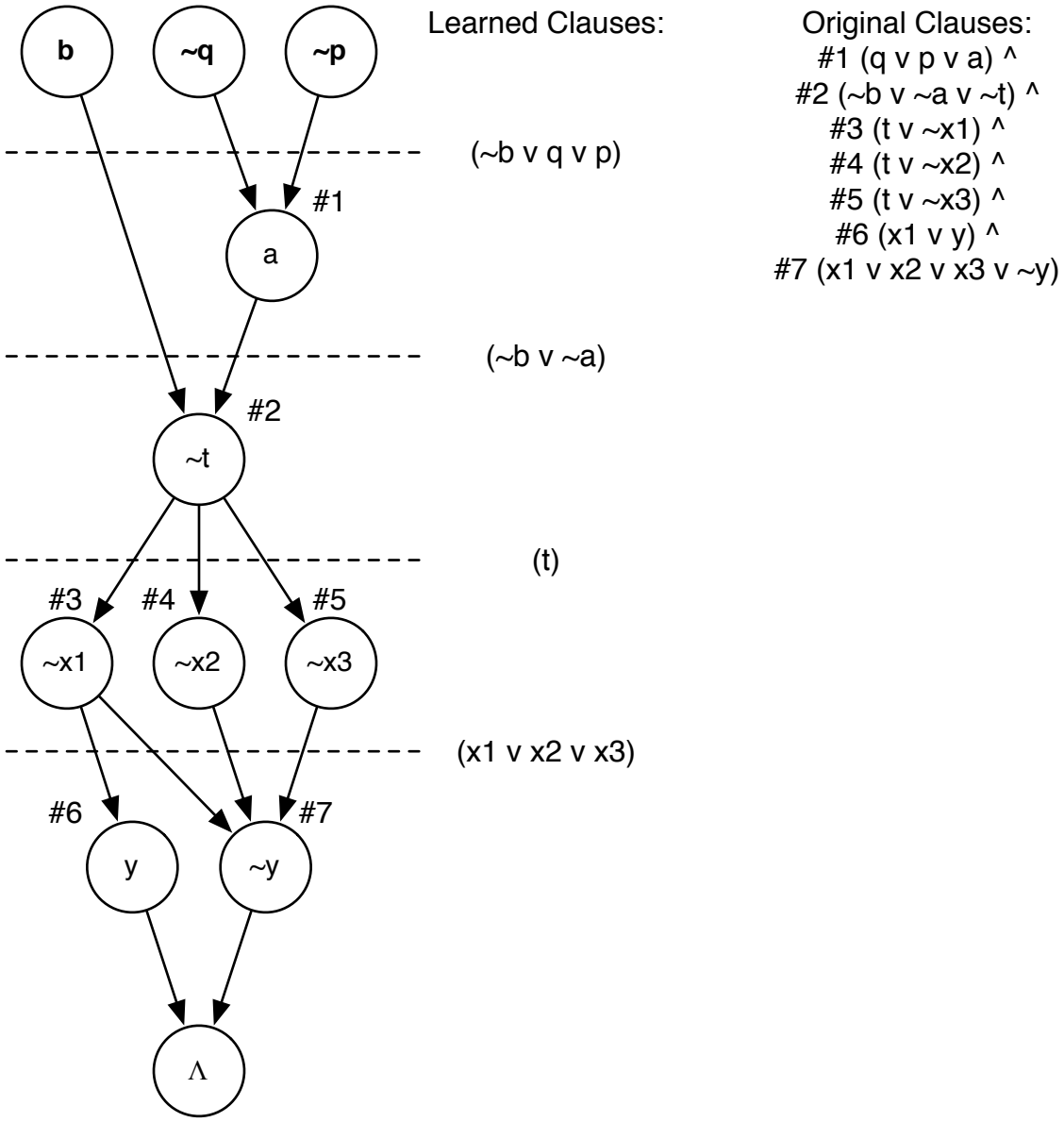


Figure 1: Conflict graph with clauses learned from possible cuts. Adapted from Beanie [1].

literals from the decision variables; we then take the negation of the literals in the nodes which have an outgoing edge crossing the cut. This forms our new clause. The new clause is consistent with the old set of clauses, since a resolution derivation exists for this clause (see Beanie et al. [1] for proof).

The question remains: which cut of the graph do we use? We choose to use the cut farthest from Λ. This represents the *most* reasoning we could do about the state; considering that information is lost when the algorithm makes random steps, we believe this heuristic should be quite effective. Choosing a cut this far back in the tree may be disadvantageous as well; we may learn very long clauses, which will frequently not be applicable. An in-depth investigation of other graph cut heuristics would be quite valuable, e.g. shortest cut, some greedy search for the shortest cut, etc.

5 Results: Single Processor

5.1 Methodology

To validate this approach, we show that WalkSAT with clause learning outperforms regular WalkSAT, both in terms of number of steps needed and in terms of actual CPU execution time. We measured the performance (steps taken and runtime) of both versions on several standard benchmarking exercises from SATLIB, and present some of these results below. In general we found substantial improvement on larger problems, while on smaller problems the improvement tended to be below our measurement threshold. On all problems we achieved a reduction in the number of steps required to solve the problem, but on some smaller problems we were unable to obtain an improvement in runtime. The problems included below are the largest problems our algorithm was able to solve within memory and time constraints.

We present results for a blocks-world planning problem and two 2D-graph coloring problem. We also tried several smaller problems from these domains, but did not see substantial benefit in runtime from clause-learning on these smaller problems. We attribute the inability to achieve runtime improvements on smaller problems to the comparative cheapness of taking a WalkSAT step. For large problems, steps are more expensive in relation to clause learning. These results suggest that our algorithm will generate significant improvement in runtime on large SAT problems, and significant improvement in steps on all SAT problems. In cases where we have significant access to learned clauses (such as multithreaded domains), runtime should show an improvement even on small problems and an even greater improvement on large problems.

Of note is that our entire implementation, including I/O and a CNF file parser, is 600 lines of well-formatted and commented C code. We rely on no libraries other than the C standard library, and our only data structures are arrays. Our code lacks the heavily optimized data structures used in most modern SAT solvers, so the runtimes for our algorithm are not directly comparable to the runtimes for state-of-the-art solvers. They are intended for comparison between WalkSAT methods.

Each point on our graphs is the median of 100 runs. Since each set of runs contained some extreme outliers, we cannot compute accurate 95% confidence intervals using the usual Gaussian assumptions. Instead, we test the significance of our results using a Wilcoxon signed-rank sum test. This tests the hypothesis that the results came from different distributions vs. the null hypothesis that they came from the same distribution. For each graph, we vary the learning interval, which is the number of steps between learning new clauses. We varied the learning interval in powers of 2 starting from 16. Results for learning intervals faster than 16 showed further decreases in the number of steps required for a solution, but caused prohibitively higher runtimes. We present results for regular WalkSAT with the same restart interval and random step probability as a horizontal line. Note that this is equivalent to WalkSAT with an infinite clause-learning interval.

We capped runs at 500,000 steps. We varied the restart interval exponentially from every 100 steps to every 500,000 steps and determined that a restart frequency of once every 500,000 steps produced the best performance for both WalkSATs. This corresponds to no restarts for our problems.

5.2 Blocks-World

First, we examine the results from a blocks-world planning problem described by Kautz and Selman [7]. We consider the results for *bw_large.a*, which has 459 variables and 4675 clauses. Smaller

instances were solved too quickly to get meaningful measurements, and larger instances were prohibitively large for regular WalkSAT (although the clause learning version was able to solve *bw_large.b*, which has 1,087 variables and 13,772 clauses, in 224 seconds, averaged over 100 runs). For *bw_large.a*, Figure 2 shows that WalkSAT with clause learning is able to find solutions about five times faster with an appropriate clause-learning frequency (around 64 steps for every clause learned), and we achieve nearly an order of magnitude fewer steps taken with a clause learning interval of 16 steps. This observation supports our claim that, if we can leverage a low-cost source of additional clauses (such as another parallel instance of WalkSAT) we can expect further improvements in solution time.

Results are significant (Wilcoxon test has p-val less than 0.05) up to a learning interval of 8192 steps. After this point, we are only learning a handful of clauses each run, and it would take prohibitively many samples to distinguish the two algorithms. Results after this mark are included for completeness.

5.3 2D-Graph Coloring

Next, we examine the results from two typical 2D-graph coloring problems described by Tad Hogg [6] (see Figures 3 and 4). These problems are a SAT encoding of the standard graph 3-coloring problem (is a given planar graph colorable with 3 colors?). The first instance, *flat125-1.cnf*, has 125 vertices and 301 edges, and is encoded into a SAT formula with 375 variables and 1403 clauses. The second instance, *flat200-1*, has 200 vertices and 479 edges, and is encoded as a SAT formula with 600 variables and 2237 clauses.

We note that we achieve a 3x improvement in the number of steps required. In this domain, clause learning is cheaper than in the blocks-world domain, so we note that our runtime is slightly worse.

Results are significant for both problems (Wilcoxon test has p-val less than 0.05) up to a learning interval of 2048 steps. Results after this mark are included for completeness.

6 Parallelization

As computers become more and more parallelized, scaling our algorithms to work in parallel becomes increasingly important. Here, we discuss the issues with parallelizing WalkSAT and clause learning.

In a cluster environment, we have several general-purpose CPUs that are able to communicate in some fashion. By contrast, on a GPU, we have many more special-purpose CPUs that are only able to communicate through shared memory. Our algorithm can be tuned to work effectively across a range of interconnect technologies (network communication, in-memory communication, etc) by changing the clause learning interval (as discussed below). In addition, as long as throughput stays relatively constant, the algorithm is insensitive to latency.

The naive parallelization method is, if we have n CPUs, run one WalkSAT instance on each CPU. This should produce some improvement; it allows us to take the best of n runs. Intuitively, we can see that we may be able to do better by allowing communication between the processes, so they don't all work independently. We would like to share information across the processes to allow the threads exploring less promising areas of the state space to "warn away" their siblings. As described above, the method we suggest for doing this is through clause learning; we construct a shared clause database that parallel WalkSAT instances contribute their learned clauses to. This should allow us to leverage the computation performed by different threads to allow each thread to improve its performance. By sending learned clauses to all the other threads, each thread gains some benefit for having examined a region of space without having to actually visit it.

7 Results: Multiprocessor

We present results from running four algorithms on each problem described in the single-threaded section. Our first algorithm is WalkSAT with no clause learning. We expect to see some improvement for this algorithm just by increasing the number of threads. Our second algorithm is WalkSAT

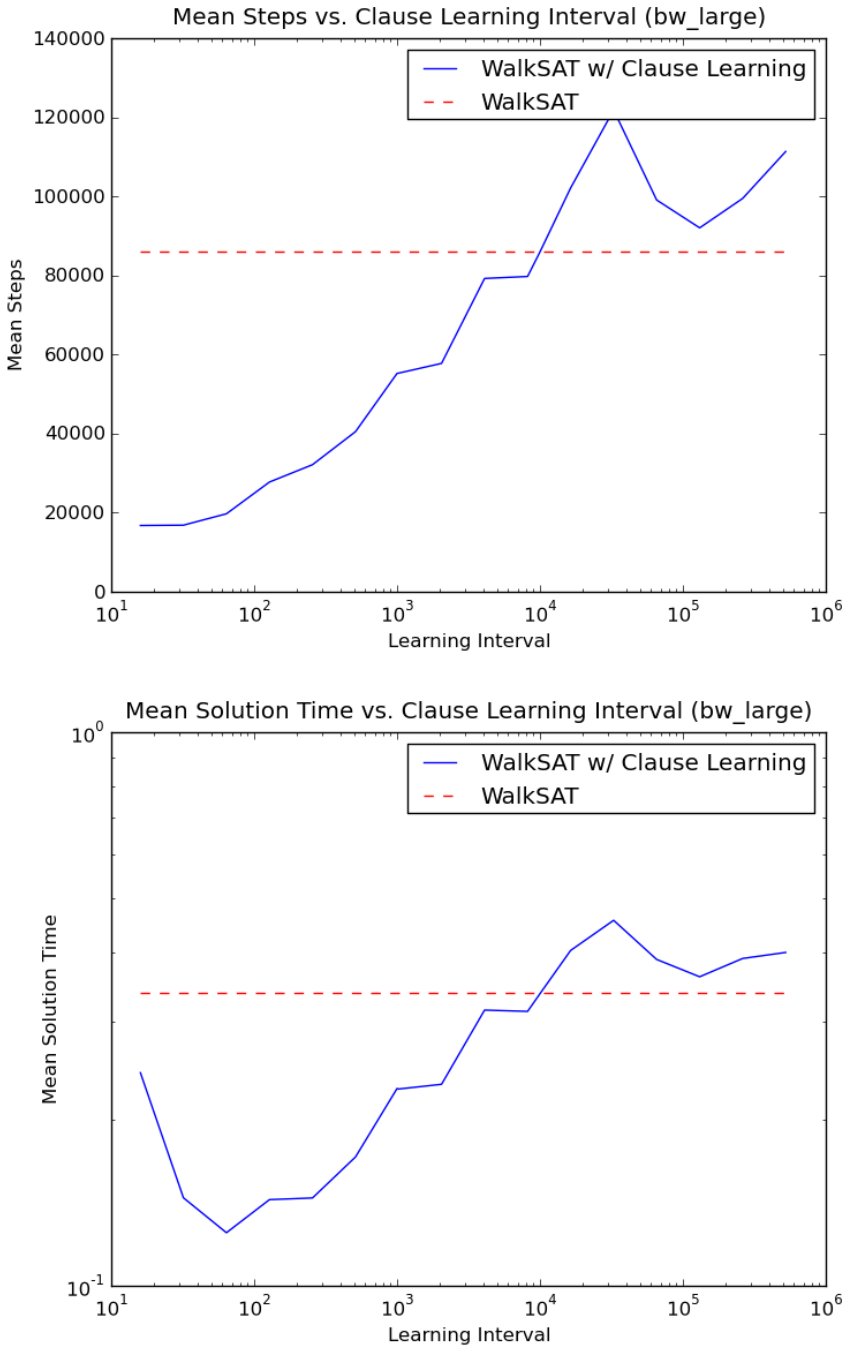


Figure 2: Steps and runtime needed for Walksat vs. Walksat with clause learning, with varying clause learning intervals. Each point is the median of 100 runs.

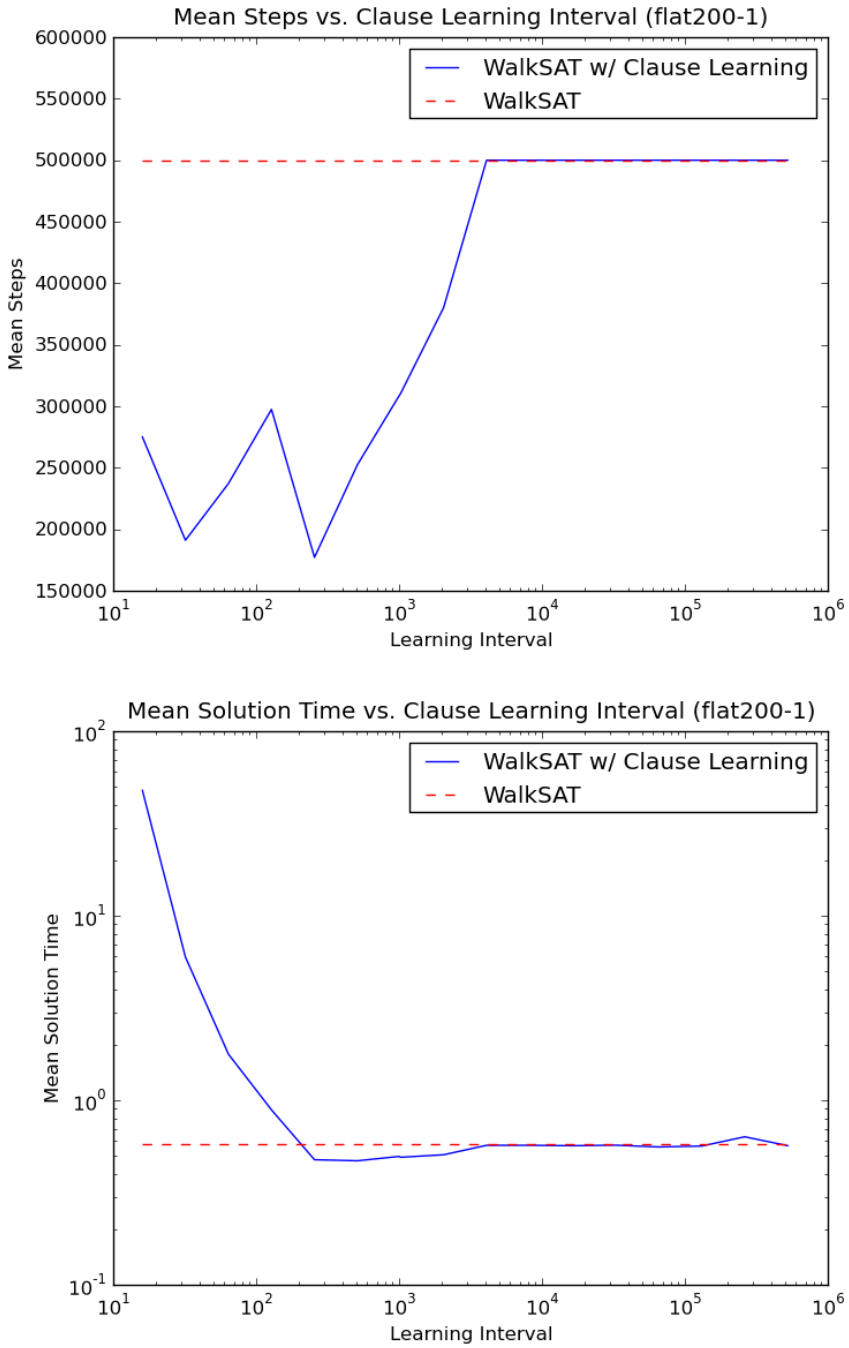


Figure 3: Steps and runtime needed for Walksat vs. Walksat with clause learning, with varying clause learning intervals. Each point is the median of 100 runs. In the top graph, the line for WalkSAT is at 500,000 on the y-axis.

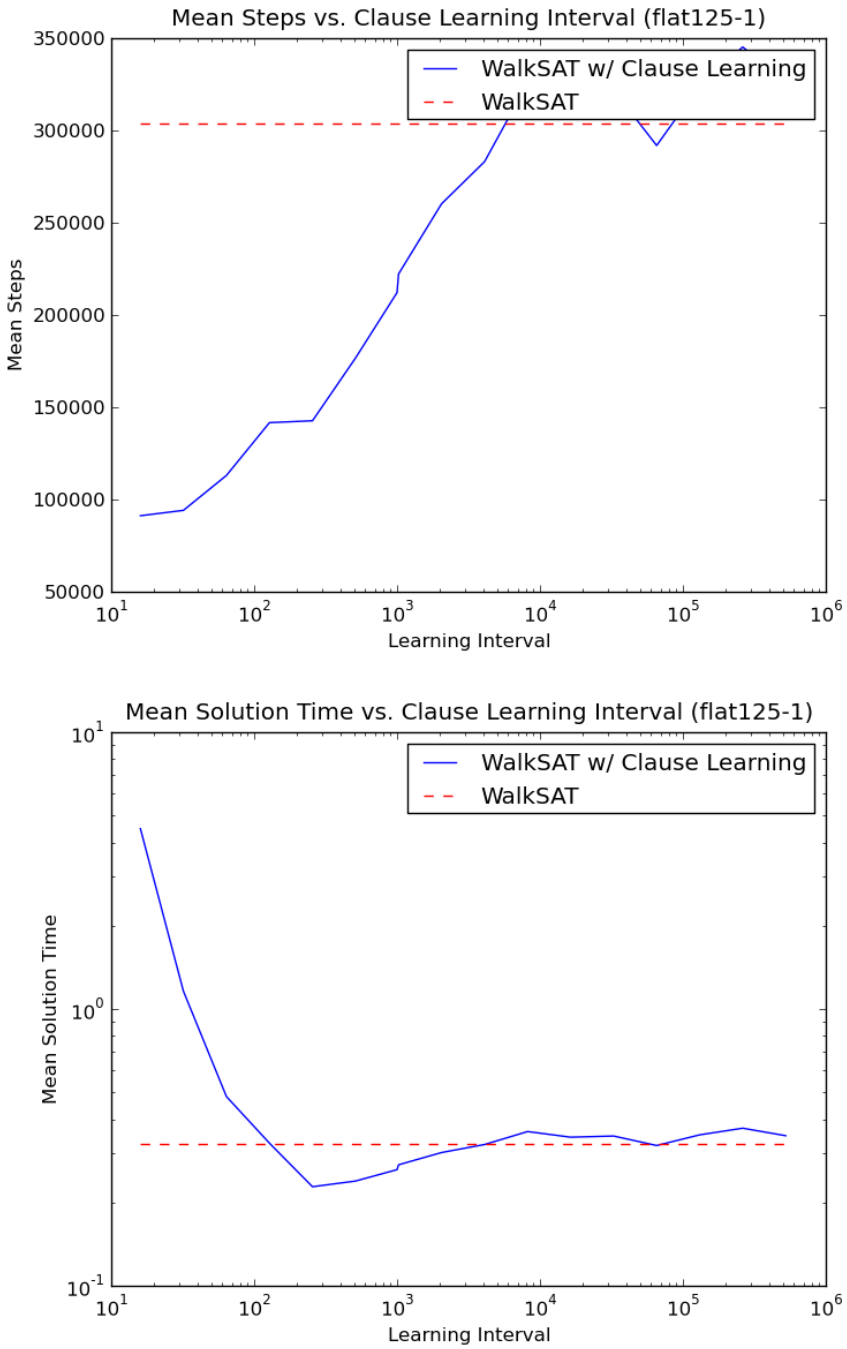


Figure 4: Steps and runtime needed for Walksat vs. Walksat with clause learning, with varying clause learning intervals. Each point is the median of 100 runs.

with clause learning, but no sharing of learned clauses. Again, we expect some improvement just for scaling the number of threads.

We also compare to WalkSAT with clause learning, where all clauses are shared among each thread. There are two natural points of comparison. One point of comparison is the when the shared clause database size is equivalent to the size of clause database for the non-sharing version. This requires us to scale down the learning interval as we increase the number of threads (so that each thread learns a clause every $i * n$ steps, where i is the base learning interval for the non-sharing version and n is the number of threads). This allows us to compare the quality of the shared database more easily. The other natural point is to allow all threads to learn at the same rate. This will produce a shared clause database that is n times larger than the database the non-sharing threads have access to.

To control for threading issues, we simulate our multithreading by iteratively polling each thread. We only count the runtime each thread spends executing its code. We used a learning interval of 1000 steps for each algorithm.

To determine results significance, we perform a Wilcoxon test pairwise between the results for each algorithm. Unless noted, all results are significant with a p-val of .05.

We conclude by observing that for large SAT problems, we are able to obtain significant performance increases. For some problems, such as the 2D Graph Coloring problems, sharing is a noticeable improvement; however, clauses from other threads seem to be less useful than the ones generated by a thread for itself. Finally, the results from random 3-SAT suggest that the comparative expense of making a WalkSAT step vs. learning a clause is an important part of achieving better runtimes.

7.1 Blocks-World

In Figure 5 we present results for the *bw_large.a* problem described above. We observe that WalkSAT with clause sharing, with no reduction in the learning rate, outperforms the other 3 algorithms by a significant margin. WalkSAT with learning, and WalkSAT with sharing but a reduced learning rate performed about equally, and in fact were indistinguishable to our Wilcoxon test. Plain WalkSAT performs the worst.

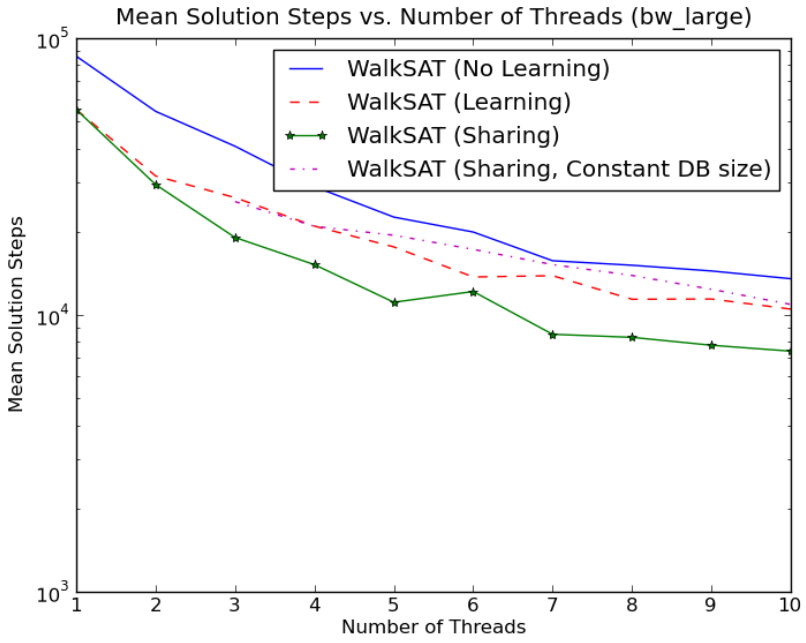
7.2 2D Graph Coloring

In Figures 7 and 6 we present results for the *flat200-1* and *flat125-1* problem described above. Here we note that WalkSAT without sharing and WalkSAT with sharing perform about the same (and are indistinguishable to the Wilcoxon test). WalkSAT with sharing, but a reduced learning rate performs noticeably worse. We suggest that this is due to clauses not translating well between threads. It appears that, in this problem domain, clauses learned by threads at other locations in the problem space are significantly less helpful than the clauses a thread learns by itself. However, the thread spent zero time learning the other clauses.

8 Graphics Processing Units

Recently, chip manufacturers such as NVIDIA and ATI have begun opening their graphics processing devices up to general-purpose computing, even producing computing-specialized versions of these devices. This is an exciting development; an NVIDIA Tesla C1060 retails for around $1300.00 and contains 240 cores, each clocked at 1.3 GHz, and is capable of 933 GFLOPs (billion floating point operations per second). In comparison, an Intel Core i7-965 Extreme Edition retails for around $1000 and is capable of about 70 GFLOPS.

However, GPUs were not originally designed for general-purpose computing, and have some strong constraints on the types of algorithms they can run. GPUs are "Single-Instruction, Multiple-Data" devices: with some caveats, algorithms running in parallel must all execute the same instruction at the same time, although their memory can hold different data. The operating paradigm is "data parallelism" as opposed to "computation parallelism" or "code parallelism." This makes control-flow-heavy algorithms like backtracking search impractical to parallelize on GPUs (and was the original motivation for this research). WalkSAT, on the other hand, contains minimal control flow.



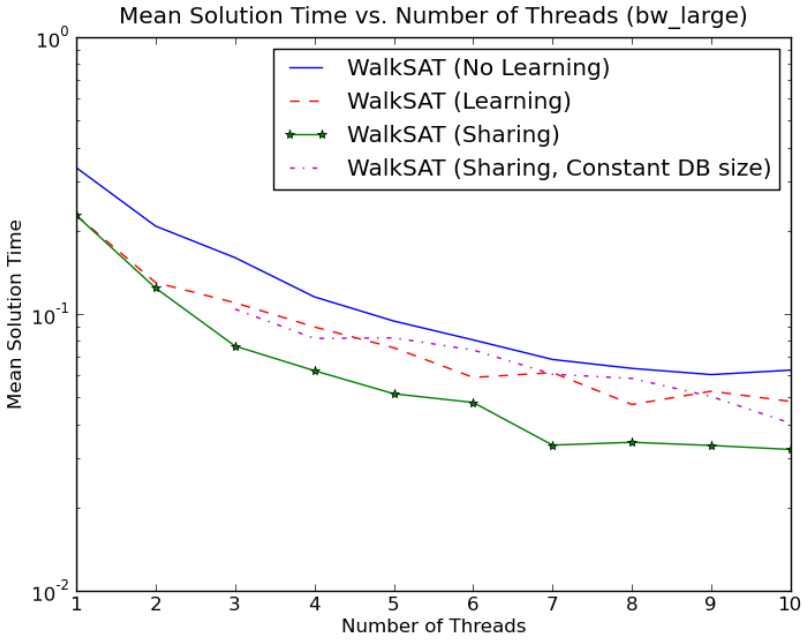
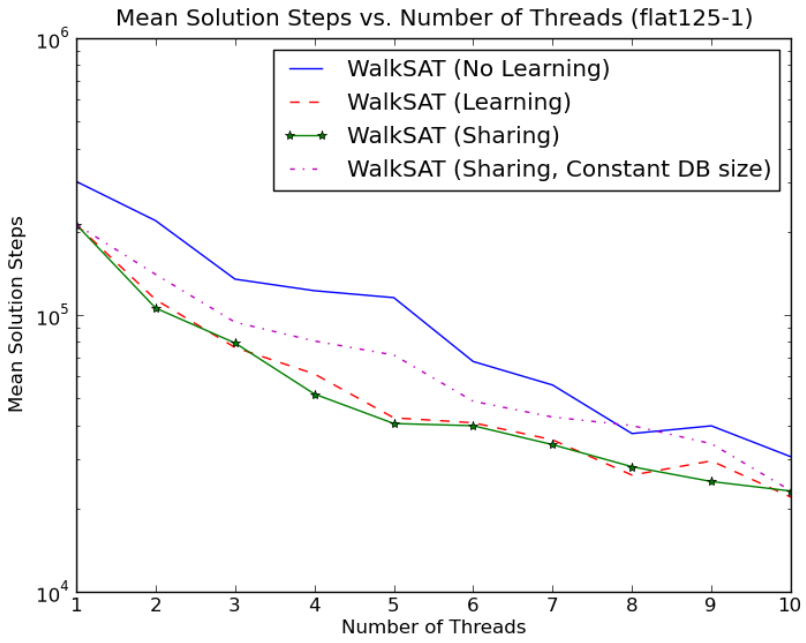


Figure 5: Steps and runtime needed for different WalkSAT versions with varying number of threads. Each point is the average of 100 runs.



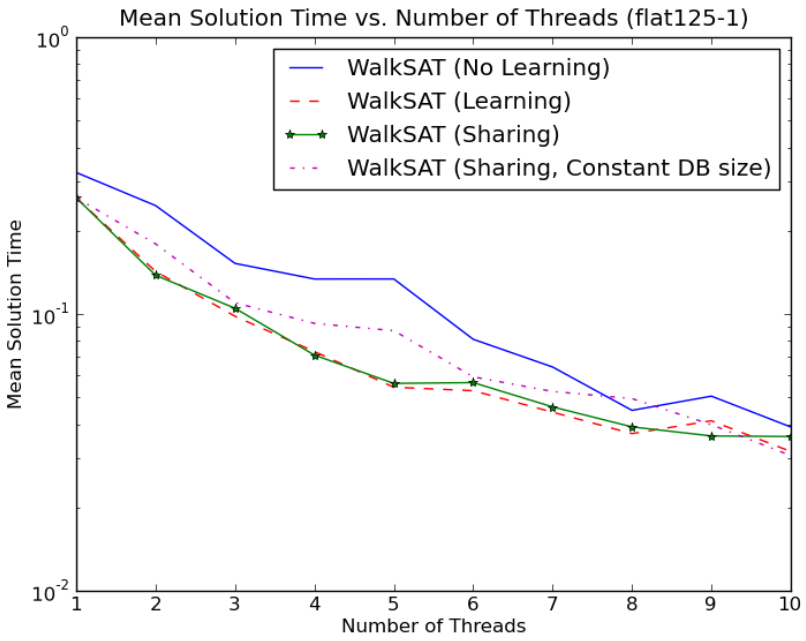
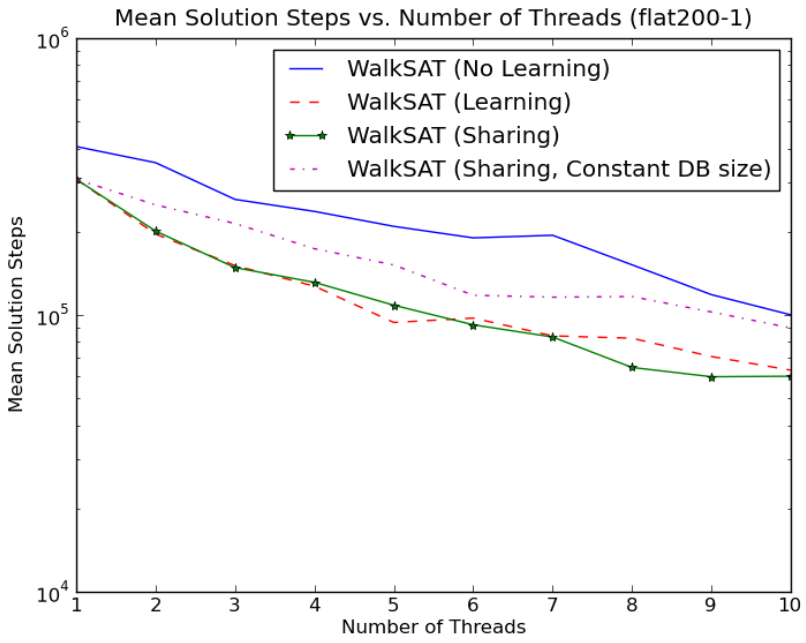


Figure 6: Steps and runtime needed for different WalkSAT versions with varying number of threads. Each point is the average of 100 runs.



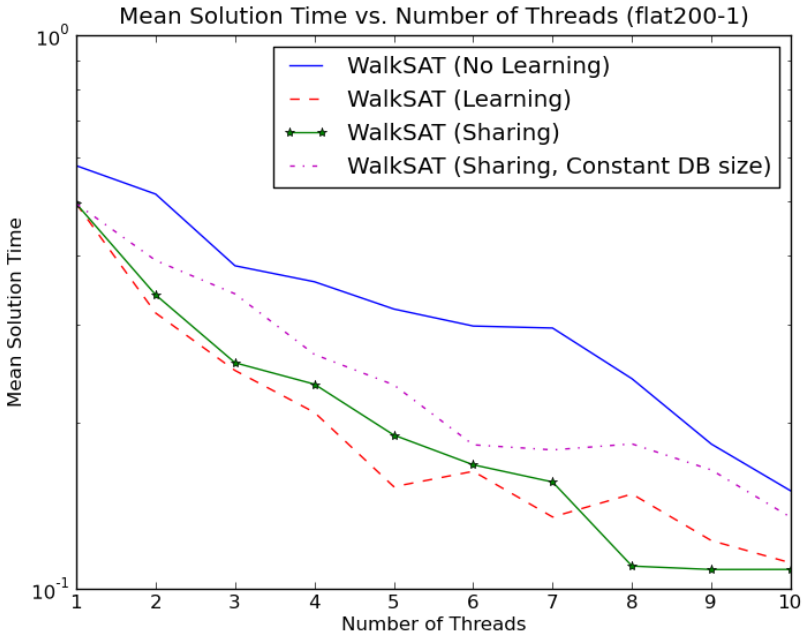


Figure 7: Steps and runtime needed for different WalkSAT versions with varying number of threads. Each point is the average of 100 runs.

We can slightly relax the requirement of lock-step execution by *masking*. GPUs use masking to instruct some subset of the running threads to not execute pieces of code (sleep while that code is run). Control flow, such as "if" statements, is implemented in this fashion: threads taking the "true" branch of an "if" statement execute that branch while the other set of threads lies dormant; then, the second set of threads takes the "false" branch while the first set lies dormant. Thus, if any thread takes a branch, all threads must wait on that branch to complete.

8.1 WalkSAT for GPUs

WalkSAT parallelized for GPUs takes the form of thousands of threads, all running an identical set of code in lock-step. They must keep some local state, which includes their own assignment, as well as which clauses are satisfied/unsatisfied, and their individual random number generator state. The clauses can optionally share a global clause database, or keep their own individual clause database. Then, we simply execute Algorithm 1 on each thread.

We need to examine the amount of control flow required for the steps in this algorithm to ensure that it utilizes processors efficiently (with a minimum of masking). We first discuss a simple version of Algorithm 1 without optimized data structures. We discuss some optimization possibilities below. An important part of future work will be to experiment with an implementation of Algorithm 1 and develop and optimize data structures specialized to the SIMD setting.

At each step, we must choose a random unsatisfied clause by calling getRandUnsatClause, which requires scanning through the set of clauses. The only control flow is in terminating the loop; all threads must wait for all the other threads to finish this loop.

Then, most times, we expect to execute both branches of the "if" in sequence. For those threads flipping a random variable in the chosen clause, we are simply indexing a vector; the threads taking maximization steps must wait. Then, we execute the maximization step; threads taking random steps must wait. For the maximization step we must call getBestVarToFlip. This function loops through the chosen unsatisfied clause and calculates the number of clauses that would be satisfied by flipping that variable, which requires a complete scan (with no early termination). The outer loop, which iterates through the clause variables, can terminate early. All threads must wait for the thread with the longest clause to finish scanning. After this, we have a simple assignment and can continue the loop.

The places we lose the most efficiency are in getRandUnsatClause, when some threads take much longer to reach their random unsatisfied clause than the others; and in getBestVarToFlip, when some threads have longer unsatisfied clauses than others. One option for mitigating the efficiency loss in getRandUnsatClause would be to maintain a vector of unsatisfied clauses. This would allow us to simply index the vector. However, this vector might require significant updating at the end of the loop for steps involving variables in large numbers of clauses. This updating may require a good deal of control flow; which method would be faster on a GPU would depend heavily on the vector implementation. Unfortunately, for getBestVarToFlip, it is unclear how one could circumvent the requirement of having to check every variable in the clause; the "watch literal" trick described above may help with this. The inner loop, which calculates the number of unsatisfied clauses for that variable setting, could be sped up by using some of the optimizations discussed in Section 3.

Generating pseudo-random numbers is an important piece of our algorithm. Typically, random number generation involves keeping some state and applying an arithmetic transform to it to produce a pseudo-random number and a new state. This should require no masking. However, it can be quite expensive. If we are willing to sacrifice the independence of our threads, we can simply choose one branch of the outer "if" at random for all threads to follow; this will improve the speed of the threads at the cost of possibly harming their searches. This will not only save some random number generation, but remove a large source of thread inefficiency. It is difficult to estimate what effect (if any) this dependency among the threads would have. We can further save some random number generation by not probabilistically choosing branches at all, and simply having threads alternate which path they take (in step) according to some predetermined scheme.

We could further cut down on the expense of random number generation by having other threads generate random numbers for the WalkSAT threads, since this can be done independently of the other calculations. This could be executed by the main CPU or other threads on the GPU. If it becomes

too difficult for other threads to keep up, it would be possible to simply generate one random number per step and have each thread hash it with a unique, per-thread value. The hash function would need to be simple, e.g. XOR, to result in any real gains. This would come at the cost of threads no longer independently choosing clauses or variables; however, the independence lost in this fashion should be negligible.

Incorporating clause learning into this parallel WalkSAT version can be done in two ways. The simplest way would be to have each WalkSAT thread perform clause learning in-step. Unfortunately, the clause learning algorithm is very control-flow heavy and requires keeping complicated data structures, neither of which is good for GPU programming. Instead, we suggest using other threads to learn clauses from the state of a WalkSAT thread; these threads could be run on one of the main system CPUs, since typical general-purpose processors have little problem with control-flow-heavy code. The GPU runs simple parallel WalkSAT threads; the CPU peeks at the state of a WalkSAT thread and learns a clause for it, and then sends the new clause either to a central clause repository or to the thread it learned the clause from (or some variation on this scheme).

9 Future Work

A number of interesting modifications to this algorithm are worth investigating. What effect does the particular cut of the graph have on the learned clauses, and what is the most effective strategy for choosing cuts? Also, when adding large numbers of clauses, some of the clauses will invariably be more effective than others. We notice that in the graph coloring problems above clauses from other threads seem to be less useful than in the blocks world planning problems. Can we achieve any gains by pruning unused clauses? Can we develop some way to measure the quality of learned clauses, or the quality of a source of learned clauses?

This simple algorithm is not yet competitive with state-of-the-art SAT solvers like Minisat [14] (by several orders of magnitude). Some of this performance difference can be attributed to the heavily optimized data structures used by Minisat; Minisat is able to perform several orders of magnitude more variable flips per second than our implementation of WalkSAT. Can we adapt the data structures used by Minisat to work with WalkSAT, especially in a parallel environment? Of course, not all of the improvement can be attributed to more efficient data structures; Minisat uses may advanced strategies for guiding its search. Can we adapt some of the strategies used by Minisat and similar solvers, in particular conflict analysis techniques, to work with WalkSAT?

The clause learning algorithm itself is currently very difficult to parallelize, and must be run on a standard CPU to achieve acceptable performance. Is it possible to adapt the clause learning algorithm to minimize the control flow, to render it more easily parallelizable to GPUs?

Finally, we think this algorithm is a vital step towards a better stochastic local-search MaxSAT solver or a 0-1 integer linear programming solver. Extending this algorithm to the MaxSAT domain would be quite valuable.

10 Conclusion

In conclusion, we have presented an extension to the basic WalkSAT algorithm that achieves significant gains in speed, both in terms of number of WalkSAT steps and run time. We observe that problems where taking a WalkSAT step is comparatively expensive benefit more from clause learning than ones where a WalkSAT step is relatively cheap. We then show that our implementation derives significant improvement from running on multiple processors. We note that, in some problem domains, the quality of a clause learned from another thread is less than the quality of a clause learned by the thread itself. Next, we analyzed our algorithm and suggest that it is a promising candidate for implementation on simpler, highly parallel processors like GPUs. Finally, we discuss areas for further exploration.

References

[1] Paul Beanie, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages

1194–1201, 2003.

[2] Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2:299–306, 1997.

[3] Marco Botta, Attilio Giordana, Lorenza Saitta, Michle Sebag, James Cussens, and Alan M. Frisch. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

[4] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proc. IJCAI-91*, pages 331–337, 1991.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM*, 5:394–397, 1962.

[6] Tad Hogg. Refining the phase transition in combinatorial search. *Artificial Inteligence*, 81:127–154, 1996.

[7] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[8] Henry Kautz, Bart Selman, and Jerg Hoffmann. Satplan: Planning as satisfiability. In *in International Planning Competition at the 14th International Conference on Automated Planning and Scheduling. [Online]. Available: http://www.cs.washington.edu/homes/kautz/satplan*, 2004.

[9] Matthew Richardson and Pedro Domingos. Markov logic networks. In *Machine Learning*, page 2006, 2006.

[10] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.

[11] L. Stockmeyer. On approximation algorithms for #p. *SIAM J. Computing*, 14:849–861, 1985.

[12] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*, pages 115–125. Steklov Mathematical Institute, 1968.

[13] Miroslav N. Velev. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Journal of Symbolic Computation*, pages 226–231, 2001.

[14] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *In ICCAD*, pages 279–285, 2001.