Parallelizing MiniSat

6.884 Final Project Report

I-Ting Angelina Lee
angelee@mit.edu

Zhunping Zhang
zhunping@csail.mit.edu

ABSTRACT

The ***Boolean satisfiability (SAT) problem*** asks whether the variables in a given Boolean formula can be assigned in such a way that the formula evaluates to true. Given a Boolean formula, a SAT solver tries to find a set of satisfying assignments or declares that the formula is unsatisfiable. In this project, we implemented a parallel SAT solver using the Cilk multithreaded programming language. Our parallel implementation is based on MiniSat, a well regarded serial SAT solver known for its efficient implementation and small code base. This report outlines our parallelizing strategy and documents the implementation details of the search function. At the end of the document, we also presents some preliminary experimental results. The parallelized MiniSat has performance comparable to the original implementation when running on single processor. When running with multiple processors, however, the throughput of the solver does not quite scale as the number of processors increases.

1. Introduction

The ***Boolean satisfiability (SAT) problem*** is a classic NP-complete problem [6], which asks whether the variables in a given Boolean formula can be assigned in such a way that the formula evaluates to true. Typically, the input Boolean formula is expressed in ***conjunctive normal form (CNF)***, which is a conjunction (`AND`) of ***clauses*** composed of disjunction (`OR`) of literals, where a ***literal*** is either a variable or the negation of a variable.

A SAT solver has many useful applications such as Electronic Design Automation [13], program verification [8], and constraint solving in artificial intelligence. Consequently, researcher spent considerable amount of effort developing heuristics to organize and prune the search space, thereby allowing larger problem instances to be solved. Recent progress made in SAT solving includes techniques such as ”restarting” [9], “conflict-driven backtracking” [12], and “dynamic variable ordering” [14].

Another ongoing trend in SAT solving is to harness the multicore architecture to search in parallel. The majority of the parallel solvers are based on a parallel divide and conquer method called “guided path” [16]. The ***guided path*** approach divides the search space into disjoint subspaces by branching on each “assumption” variables, and such subspaces can be searched in parallel. As threads traverse the subspaces in parallel, they accumulate information useful for other threads. The information may be shared through either a centralized database with thread-local references [11] or a distributed database that is periodically synchronized [4, 15]. Yet a completely different approach to parallelize a SAT solver is to run multiple independent serial solvers with different “tuning parameters” for heuristics and terminates when one of the subsolvers find a result [7].

In this project, we implemented a parallel SAT solver based on MiniSat [3], which is a well regarded serial SAT solver known for its efficient implementation and small code base. The original MiniSat implementation employs advanced heuristics such as restarting, conflict-driven backtracking, and dynamic variable ordering, and its intricate structure makes it challenging to parallelize. We parallelized MiniSat using the Cilk-5 multithreaded programming language [5] and kept its structure with strategy similar to the guided path approach. We as well allow threads to share information by having each thread maintaining its own local database that periodically synchronizes with the centralized database.

While our parallel SAT solver seems to function correctly, the preliminary experimental results show that the throughput of the solver does not scale well as the number of processors increases. This is not too surprising, given that our implementation is not highly optimized, and we have not spent much time to investigate in performance issues. Nonetheless, we present this preliminary results to give the readers a sense of where our implementation stands.

The rest of the report is organized as following: Section 2 gives an overview of the original MiniSat implementation to provide some background. Section 3 introduces Cilk-5’s linguistics and execution model, and outlines our parallelizing strategy using Cilk. Section 4 provides the implementation details and describes some subtle issues we encounter during implementation. Section 5 gives the preliminary results of our current implementation. Finally Section 6 concludes with some topics of future investigation.

2. Serial MiniSat overview

In this section, we go through some background necessary to explain our parallel SAT solver implementation. First, we introduce some standard SAT solver terminologies that we use throughout the rest of this document. Next, we give an overview on how the serial MiniSat works, focusing on two particular strategies used by MiniSat, the conflict-driven backtracking and the dynamic variable ordering. This section is meant to give a high level overview of MiniSat with enough details to convey how we implemented the parallel MiniSat; it is not meant to serve as a complete tutorial on the implementation of MiniSat. We refer interested readers to [3] for more details.

Before we introduce MiniSat, we need to first go over some terminologies in SAT solver used throughout the rest of this document. Given a Boolean formula expressed in conjunctive normal form, a SAT solver decides whether the variables in the formula can be assigned in such a way that the formula evaluates to true. A Boolean variable can take the value of true, false, or being free. Similarly, a literal can take the value of true, false, and free. Given a set of variable assignments, a clause is ***true*** if it contains at least one true literal; a clause is ***false*** if it contains all false literals. On the other hand, a clause is ***asserting*** if it contains one free literal and the rest of its literals are false. When a clause is asserting, a solver would try to ***assert*** the free literal, *i.e.*, assign value true to the free literal, so that the clause does not become false and render the entire formula false. We refer to the last free literal in an asserting clause as the ***asserting literal***. Finally, a clause is ***free*** if it is not true, not false, and not asserting.

One can think of the search space as a binary tree, where each node represents a variable to be assigned next, and the two outgoing edges leading to the its children represent the true and false assignments to the variable. A naive implementation of a SAT solver may perform an exhaustive enumeration of assignments by traversing through the search space in a depth-first fashion, fixing the variable assignments one by one; when it reaches a ***conflict***, *i.e.*, a variable assignment that falsifies the formula, it backtracks to the previous level, reassigns the conflicting variable to a different polarity, and tries again.

MiniSat as well performs a exhaustive enumeration of assignments, but it employs few techniques to speed up the search process. One of them is referred as the "conflict-driven backtracking." In MiniSat, there are two different means for a variable to be assigned. When MiniSat chooses a free variable and assigns a value to it, the assignment is called an ***assumption***, or the variable is ***assumed***. The number of assumptions made thus far defines the ***level*** of the solver. Once an assumption is made, some clauses in the database may become asserting, and solver must assign the asserting literals true in order to continue the search. The assignment to the asserting literals may further cause more clauses to be asserting and more variables to be assigned. Such a cascading process of assigning variables is referred as ***propagation***, or a variable is ***propagated*** if its assignment is forced due to a propagation. MiniSat keeps a list of assumptions made and a log of all variable assignments made in chronological order during the search process. When an asserting literal is assigned to true, MiniSat as well remembers the ***reason*** as to why this assignment is made, *i.e.*, the asserting clause that asserts the literal.

The process of propagation stops either when there are no more asserting clauses, or when the solver encounters a conflict, where the propagation attempts to assign true to an asserting literal, but its corresponding variable has already been assigned, and the assignment makes the asserting literal false. At this point, the search process is stuck, and the solver must backtrack. Before the solver backtracks, it examines the clause that asserts the conflicting value, and traverses the reason backwards to figure out why such conflicting assignments are made. The solver then generates a ***learned clause*** to represent the conflict, and its clause database is expanded with the learned clause to prevent such conflicting assignments and help pruning subspaces in the future search. The conflict analysis also helps the solver to figure out the level which it should backtrack to in order to resolve the conflict. Upon a conflict, instead of reverting one variable assignment at a time, the solver may cancel multiple levels of assignments. This process of analyzing the conflict and backtracking multiple levels until conflict is resolved is referred as the ***conflict-driven backtracking***.

In order to facilitate the propagation and efficiently figure out which clause is asserting, MiniSat also keeps tracks of a list of ***watched literals***, where two free literals are selected from every clause at the beginning of the search. Whenever a watched literal in a clause is assigned with value false, the solver swaps the watched literal with another literal from the same clause that is either true or still free. The solver can then quickly determine whether a clause is asserting, *i.e.*, when the solver cannot find a replacement for a falsified watched literal from the same clause.

MiniSat also spend considerable amount of effort to order its variables in order to determine which variable to assume next. It can be observed empirically that, the choice of ordering has profound influence on the performance. During the search, MiniSat keeps track of an array of values associated with each variable, referred as the ***activity***. The activities of variables are as well conflict driven: during conflict analysis, the solver increase the activities of all variables encountered during the process of traversing the reason backwards. The increment of the activity is assigned in such a way that favors the more recent conflicts than the less recent ones. The variables are kept in a max heap called ***order***, which is sorted at all time, and the variable with the highest activity is selected to assume next. This heuristic used to select variables is referred as the ***dynamic variable ordering***.

3. Parallelizing MiniSat with Cilk-5

This section gives an overview on how we parallelize MiniSat with Cilk-5. In order to present our strategy, we first introduce Cilk-5's linguistic model and some basics of its work-stealing scheduler.[1] Then we describe our strategy of parallelizing MiniSat, which involves answering two questions. First, we need to know what to parallelize. Second, based on the strategy of parallelization, we need to figure out how to handle various data structures used by the solver. We describe them in turn.

Cilk-5

Cilk-5 is a fork-join programming language, which permits dynamic creation of parallelism. More specifically, Cilk-5's linguistic constructs allow the programmer to denote the logical parallelism of the program rather than the actual parallelism at execution time. The Cilk-5 work-stealing scheduler [1, 2] respects the logical parallelism specified by the programmer while guaranteeing that programs take full advantage of the processors available at runtime.

Cilk-5 extends C with five keywords: `spawn`, `sync`, `cilk`, `inlet`, and `abort`. Parallelism is created using the keyword `spawn`. When a function call is preceded by the keyword `spawn`, the ***child*** function is ***spawned*** and the scheduler may continue to execute the continuation of the ***parent*** (*i.e.* caller) in parallel with the spawned child without waiting for it to return. The complement of `spawn` is the keyword `sync`, which acts as a local barrier and joins together the parallelism forked by `spawn`. The Cilk-5 runtime system ensures that statements after a `sync` are not executed until all functions spawned before the `sync` statement have completed and returned. In Cilk-5, a ***Cilk function*** that contains the keywords `spawn` and `sync` must be spawned and cannot be called. Similarly, one can only spawn a Cilk function but not a C function. The `cilk` keyword is a function modifier so that the type system can preclude calls from C functions to Cilk functions. An `inlet` is a C function internal to a `cilk` function, which provides a mean to incorporate a return value from a spawned child into the parent frame in a more complex manner. The keyword `abort` can be used only inside an `inlet`; once executed, the `abort` statement causes all the already-spawned children to terminate. The keywords `inlet` and `abort` together provide support for speculative computation, which we employ in our implementation of parallel MiniSat.

Cilk-5's work-stealing scheduler load-balances parallel execution across the available worker threads. Cilk-5 follows the "lazy task creation" strategy of Kranz, Halstead, and Mohr [10], where the worker suspends the parent when a child is spawned and begins work on the child. Operationally, when the user code running on a worker encounters a `spawn`, it invokes the child function and suspends the parent, just as with an ordinary subroutine call, but it also places the parent frame on the bottom of a ***deque*** (double-ended queue). When the child returns, it pops the bottom of the deque and resumes the parent frame. Pushing and popping frames from the bottom of the deque is the common case, and it mirrors precisely the behavior of C or other Algol-like languages in their use of a stack.

[1] We include the materials on Cilk-5 so that this document is self-contained. For readers who are familiar with Cilk and its work-stealing algorithm can safely skip over the paragraphs under the Cilk-5 subheading.

The worker's behavior departs from ordinary serial stack execution if it runs out of work. This situation can arise if the code executed by the worker encounters a `sync`. In this case the worker becomes a ***thief***, and it attempts to steal the topmost (oldest) frame from a ***victim*** worker. Cilk-5's strategy is to choose the victim randomly, which can be shown [2, 5] to yield provably good performance. If the steal is ***successful***, the worker resumes the stolen frame.

Our strategy of parallelizing MiniSat

As we mentioned previously, MiniSat differentiates between variables that are assumed versus variables that are propagated, and each assumed variable defines a new level for the solver. With this strategy, a node in the search tree with depth d corresponds to a variable assumed at level d, and the two edges corresponds to the true and false assignments of the assumed variable. The propagation triggered by a particular assumption falls on the edge corresponding to the assumed variable assignment. When the solver backtracks to level d, it clears out all the assignments (assumed or propagated) made at level d and deeper to resolve the conflict. Once the conflict is resolved, the solver then propagates the asserting literal from the learned clause and continues the search by selecting a different variable to assume at level d.

In our implementation of the parallel MiniSat, we explore the parallelism by allowing the two branches of a given node to be searched in parallel. That is, if a worker is exploring the false branch of some assumed variable $x1$, we allow a different worker to search down the true branch of $x1$. The original MiniSat is written in such way that the search is performed iteratively. In order to allow the second branch to be explored in parallel, we rewrote the search function so that the search is performed recursively; that is, after a variable is selected to assume to be false, it spawns itself recursively to perform the search for the next level. Hence, the continuation of the search can potentially be stolen and executed in parallel. If the continuation is stolen, the thief would select the same variable to assume and continue the search with the variable assigned to true.

When a worker reaches a conflict and needs to backtrack, it aborts workers working on the subtrees along the backtracking path, because it knows that the workers working on those subtrees would eventually reach conflicts. While this abort policy is not strictly necessary for correctness, it is an optimization, saving workers from searching down unfruitful paths further. The program would still be correct without the abort policy, since the workers working along the backtracking path should eventually reach conflicts and backtrack.

We chose to adapt the abort policy, and a worker backtracking needs to have a mean to abort other workers working along the backtracking path. Since the search function recursively spawns itself for each level of assume, aborting workers working along the backtracking path is equivalent to aborting subcomputations along the way as the backtracking worker returns back to the appropriate level of recursion. Therefore, we simply marks each recursion with its depth, which corresponds to the level of assume; when a worker backtracks, it returns and triggers abort along the way, until it reaches the frame whose marked depth is equivalent to the backtracking level.

How to handle various data structures

Once we parallelize the search function, we must also consider how we handle variable data structures used by the solver. We summarize a list of data structures employed by the original MiniSat below:

- `assume`: A log of variable assumed in chronological order.
- `trail`: A log of variable assignments made in chronological order, including variables assumed and propagated.
- `clause_DB`: A list of clauses, including input constraints and learned clauses.
- `assign`: The current assignment of variables.
- `reason`: The reason for each assignment made.
- `order`: The current order of variables sorted by their activity values.
- `activity`: The current activity of all variables.

The main challenge is to figure out, when a branch is stolen and searched in parallel, how many data should the thief copy over from the victim, what can be kept separately and what needs to be synchronized. At first glance, it seems that data structures such as `assume`, `trail`, `assign`, and `reason` are very much context dependent, *i.e.* the values stored in them depend on where the worker is in the search tree. In fact, given our strategy for parallelizing MiniSat, we must ensure that when a steal occurs, the thief share the same prefix of `assume` as its victim, so that the thief operates on data which correctly reflects where it is exploring in the search space. Doing so also ensures that the level of recursion always aligns with the level each assume is made, which is crucial for the abort to work correctly.

We made the observation that, as long as the thief makes a copy of the `assume` from the victim, up to the level where the work is stolen, the thief can reconstruct the rest of the data structures independently by replaying the prefix of `assume` that it just copied over. Doing so avoid making copies of all the data structures upon a steal, which we speculate may be prohibitively expensive, since a moderately sized input to a SAT solver can contain at least thousands of variables. On the other hand, we do make a trade off that now we have the additional overhead for replaying the `assume`. As it turns out that the strategy of making copies has other issues (which we describe in more details in Section 6), we chose the strategy to only copy over the `assume` and perform replay.

Therefore, we end up keeping P copies of solvers, one for each worker, and each solver contains its own copy of the data structures listed above. Upon a steal, the thief replays `assume` to bring its solver "up-to-date", so that the solver data structures reflect the stolen work that it is about to resume; this includes values stored in `trail`, `assign` and `reason`. The `order` and `activity` are tightly coupled, but they only influence which variable the thief picks to assume next, so it is less context-dependent. The only restriction is that, any variable that is still free must appear in the `order` list, which we also ensure in the replay.

The `clause_DB` seemed like it can be shared among workers as a global data structure at first. It turns out that, the original MiniSat maintains the invariant that, it always keeps the watched literals of a given clause as the first and second literals of the clause, so the solver actually manipulates the clauses stored in the `clause_DB` throughout execution. In the parallel version, since each worker explores different search path, each worker would assume and propagate different variables and therefore manipulate the clauses differently. To get around this issue, we chose the simplest strategy for now, which is to allow each worker to have its own local `clause_DB` as part of the solver state.

We don't want to give up the effect of learning, however. If one worker explores down a path, reaches a conflict, and learns a clause, we want to allow other workers to access that learned clause and gain the same knowledge as to what search space can be pruned. Therefore, we as well keep a `global_DB`. Whenever a worker learns a clause due to a conflict, it deposits the learned clause into the `global_DB`. Periodically, each worker checks with the `global_DB` and fetches new clauses deposited by other workers since it last checked. Doing so we allow workers to shared learned clauses. The `global_DB` is a shared data structure and thus needs to

```
 1  int search(Solver *solver) {
 2    int blevel, res;
 3    while(res == UNDEF) {
 4      confl = propagation(solver);
 5      if(confl) {
 6        if(level(solver)==solver->root_level) {
 7          res = UNSAT;
 8          break;
 9        }
10        blevel = analyze(solver,confl,&learned);

11        expand_DB(solver,learned);
12        cancel_until(solver,blevel);
13      } else {
14        next = select_next_var(solver);
15        if(next == UNDEF) {
16          res = SAT;
17          break;
18        }
19        assume(solver,lit_neg(next));
20      }
21    }
22    return res;
23  }
```

Figure 1: The pseudo code for the iterative search function from the original MiniSat.

be synchronized upon access. At the moment we synchronize the data structure with a reader / writer lock; we imagine that accessing the `global_DB` would not be a bottleneck, because learning does not occur too frequently. We may improve the implementation in the future if synchronizing the `global_DB` turns out to be a bottleneck.

4. Implementation details

In this section, we go into more details of the implementation by walking through the pseudo code of our parallel search function. First, we describe the serial search function implemented in the original MiniSat. Next, we show the code transformation on the search function so that the search is performed recursively instead of iteratively. Finally, we demonstrate how we parallelize the recursive search function in Cilk and discuss some subtleties that we encountered.

Figure 1 shows the pseudo code for the search function from the original MiniSat, which performs a depth-first search in an iterative fashion, terminates and returns only when a satisfying assignment has been found or when the solver declares that the formula is not satisfiable. Within each iteration, the solver first checks whether any variable needs to be propagated (line 4); the propagation may results in a conflict or it may not. When a conflict occurs (shown been lines 5–12), the solver checks whether it is at the root level (i.e. no variable has been assumed yet). If so, since a root-level conflict indicates that this formula is not satisfiable, the solver breaks out of the loop and returns `UNSAT`. On the other hand, if the conflict did not occur at root level, the solver invokes the function to analyze the conflict (line 10), which returns a backtracking level `blevel` and fills the learned clause `learned`. The solver then expends its `clause_DB` with the learned clause (line 11), and calls `cancel_until` to clear out any assignments made at `blevel` and deeper (line 12). This is equivalent to the solver backtracking to `blevel`.

If the propagation finishes without a conflict (lines 14–19), it selects the next variable according to the `order` sorted using `activity` (line 14), assumes the variable to be false (line 19), and continues the search (*i.e.*, repeats this process in the next loop iteration). If the `select_next_var` returns `UNDEF` (line 20), however, it means that there is no more variables left to be assigned, which indicates to the solver that a ***model***, or a set of satisfying assignments has been found. In which case, the solver breaks out of the loop and returns `SAT`.

```
 1  int search(Solver *solver,int depth) {
 2    int blevel = INT_INF;
 3    bool backtrack = false;
 4    while(!backtrack) {
 5      blevel = INT_INF;
 6      confl = propagation(solver);
 7      if(confl) {
 8        backtrack = true;
 9        if(level(solver)==solver->root_level) {
10          set_res(UNSAT);
11          blevel = solver->root_level - 1;
12        } else {
13          blevel = analyze(solver,confl,&learned
                 );
14          expand_DB(solver,learned);
15          cancel_until(solver,blevel);
16        }
17      } else {
18        next = select_next_var(solver);
19        if(next == UNDEF) {
20          set_res(SAT);
21          blevel = solver->root_level - 1;
22          backtrack = true;
23        } else {
24          assume(lit_neg(next));
25          blevel = search(depth+1);
26          backtrack = (blevel < depth);
27        }
28      }
29    }
30    return blevel;
31  }
```

Figure 2: The pseudo code for the serial recursive search function.

Notice that in this code, the search function takes in a parameter `solver` and passes it to the other functions. The `solver` variable is a `struct` containing data structures used by the solver as described in Section 3.

Since the iterative search is difficult to parallelize, and our parallel search strategy requires each level of assume to be its own level or recursion, the first thing we do is to transform the iterative search into a recursive search. The transformed code is shown in Figure 2. The code structure in the recursive version follows closely to the iterative one, except for the following main differences.

First in the recursive version, the function header is different from the iterative one (line 1); it takes in an additional parameter `depth`, which is equivalent to the depth of the recursion and shall as well correspond to the level of `assume`. Furthermore, the function no longer returns `SAT/UNSAT`; rather it returns a backtracking level (line 30). This is because in the recursive version, when the solver backtracks, it not only needs to perform `cancel_until` up to to `blevel`, it must as well return back to the recursion level where `depth` equals to `blevel`. For the very same reason, the loop condition changed as well (line 4). The iterative version only breaks out of loop and returns only when a model is found or when the solver declares the formula unsatisfiable. The recursive version breaks out of the loop and returns whenever it needs to backtrack.

Second, since we are no longer returning the result, whenever a root level conflict is encountered (line 9) or whenever a model is found (line 19), the solver sets the result in a nonlocal variable, via the function `set_result` (lines 10 and 20). In addition, the solver sets `blevel` to `root_level-1` (lines 11 and 21), to indicate that the control should return all the way back to the caller who initiated the search.

In the case where a conflict occurs, the code always backtracks. In the recursive version, conflicts are handled similarly as the iterative version. One thing to note is that, the code always performs `cancel_until` before backtracking. In the case where propagation finishes without a conflict, on the other hand, it similarly performs the assume. To continue the search, however, it recursively calls itself with the second argument being `depth+1` (line 25). When

```
1  int search(Solver *s,int depth,var *assume) {
2    int blevel = INT_INF;
3    bool backtrack = false;
4    var *new_assume = NULL;
5    inlet void catch(int b) {
6      blevel = min(blevel, b);
7      abort;
8    }
9    while(!backtrack) {
10     blevel = INT_INF;
11     fetch_from_globalDB(s);
12     blevel = process_fetched_clauses(s);
13     // ...
14     confl = propagation(s);
15     if(confl) {
16       backtrack = true;
17       if(level(s) == s->root_level) {
18         set_res(UNSAT);
19         blevel = s->root_level - 1;
20       } else {
21         blevel = analyze(s,confl,&learned);
22         post_to_globalDB(learned);
23         expand_DB(s,learned);
24         cancel_until(s,blevel);
25       }
26     } else {
27       next = select_next_var(s);
28       if(next == UNDEF) {
29         set_res(SAT);
30         blevel = s->root_level - 1;
31         backtrack = true;
32       } else {
33         assume(s,lit_neg(next),assume);
34         catch(spawn search(s,depth+1,assume));
35         if( !SYNCHED && blevel == INT_INF ) {
36           s = get_current_solver();
37           replay(s,assume,new_assume,depth);
38           // ...
39           assume(s,next,new_assume);
40           catch(spawn search(s,depth+1,
41                              new_assume));
42         }
43         sync;
44         if(blevel == INT_INF) {
45            break;
46         }
47         backtrack = (blevel < depth);
48         if(!SYNCHED && !backtrack) {
49           s = get_current_solver();
50           replay(s,assume,new_assume,depth);
51         }
52       }
53     }
54   }
55   if(new_assume) free(new_assume);
56   return blevel;
57 }
```

Figure 3: The pseudo code for the parallel recursive search function.

the search returns, the returned blevel is examined to determine whether it should backtrack further (line 26).

Figure 3 shows the pseudo code of the parallel recursive search function. Following the same overall structure of the recursive search, the parallel search contains a top level loop that breaks and returns when the solver needs to backtrack. As we described in Section 3, our overall strategy is to allow the two different assignment of a assumed variable to be explored in parallel. When the solver backtracks from one branch, it aborts parallel subcomputations along the backtracking path. Each worker keeps track of its own solver state, and only when a successful steal occurs, the thief makes a copy of the assume from the victim and performs replay to bring its own solver up-to-date. Thus, additional parameter called assume is passed into the search function, which keeps track of the variables assumed up to to the current level (populated by the function assume). Whenever a thief resumes the continuation, a new_assume is allocated for the thief; the thief invokes replay, which copies variables from assume to new_assume up to depth, and continues execution with the new_assume. The assume array is protected by a reader / writer lock to prevent victim from modifying the assume while the thief is making a copy. Any new_assume allocated in a given function instance is deallocated at the end of the function.

At the beginning of every loop iteration, the code fetches newly learned clauses from the global_DB and process the new clauses (lines 11 and 12). Then, the code proceeds to do propagation. If any conflict occurs, it is handled similarly as in the recursive version. In addition, it posts the learned clause resulted from the conflict to the global_DB (line 22).

When there is no conflict from propagation (lines 33–52), the code first assumes the next variable to be false and spawns the search (line 34). The worker who executes the spawn suspends the parent frame and continues executing the spawned child. At this point, if the continuation of the parent frame is stolen, the thief would resume at line 35, and depending on the if condition at line 35, it may retrieve its own version of the solver (line 36), perform replay (line 37), assume the same variable to be true, and spawn the search of the second branch (line 40).

Both spawn statements are surrounded by the call to inlet catch (definition shown between lines 5–8). The only job of the inlet is to incorporate the return value from the spawned child into the parent frame and triggers abort. If one of the spawned search encounters a conflict, it returns the backtracking level (stored into blevel) and aborts the other branch. If both spawned subcomputations happen to encounter conflicts and return around the same time, the execution of the inlet is serialized (as guaranteed by the Cilk runtime), and the smaller backtracking level is kept.

The if condition at line 35 checks for whether the continuation is stolen and whether the other branch has returned. If the continuation is not stolen and is resumed by the same worker who returned from the first spawn, the second spawn is omitted; the worker simply continues execution as the serial recursive version. On the other hand, if the continuation is stolen, but the first spawn returns and aborts before the thief resumes, the second spawn is omitted as well (blevel!=INT_INF), since this frame should be aborting.

Notice that there is a race condition here — the first spawn may return and updates blevel in catch, while the continuation after the first spawn reads blevel to determine whether to perform the second spawn. We claim that this is a benign race and does not effect the correctness of the code, because abort is an optimization; even if the thief reads a stale value and ends up executing the second spawn while the frame is being aborted, the thief would realize the abort or encounter a conflict eventually.

A sync statement is places after the two spawn statement (line 43) to ensure that control does not pass sync unless all previously spawned subcomputations have returned. A frame may be aborted due to an abort triggered by its ancestor's siblings. In which case, depending on the scheduling, it is possible for a thief to resume and pass sync successfully, but none of the inlet for this particular frame got executed, resulting blevel to remain INT_INF pass sync. In which case, since this frame is aborting, the code breaks out of the loop and returns (line 45), even though the value of blevel is not really valid, doing so is okay, because the return value of this particular frame would be discarded by its ancestor. This early return is again an optimization.

After the sync statement, the code checks for whether it should backtrack more (line 47); this check causes the loop to break and returns back to the caller whose depth equals to the backtracking level, blevel. Once the control get back to that point, the code again has to perform a get_current_solver and replay to update the solver data structures, because the worker picking up the continuation after sync may not be the same worker who returned from either spawn.

There are a couple subtleties that we omit in the pseudo code in order to simplify the presentation. First, replaying the assume can run into inconsistency; that is, a thief during replay may process an assumed variable that has been already assigned in its solver state due to an earlier propagation. This can happen because each solver keeps track of its own `clause_DB`, and its set of watched literals may differ from the victim's. The propagation may have the same value as the assumption, or it may not. If it doesn't have the same value, it is a conflict. If it does have the same value, it is inconclusive whether this branch that the thief is about to resume would bear fruit or not.

If the inconsistency occurs during the replay before the second `spawn` (line 37), the thief simply gives up the work. This does not affect the correctness of the program. Assuming the inconsistency indicates that there is a conflict, the first worker would eventually discover the conflict when it fetches clauses from `global_DB` and backtrack. On the other hand, if there were no conflict, the first worker would eventually backtrack to some level greater than the current `depth` and explore the branch that the thief gives up.

If the inconsistency occurs during the replay after `sync` (line 50), however, the thief has to handle the inconsistency, because it is only worker executing on the stolen branch. If the propagation and the assumption coincide, the thief backtracks to the depth where the later assumption is selected, and selects a different var to assume instead. If the propagation and the assumption conflict, then the thief backtrack to the most recent level where the conflict goes away. Notice that backtracking is necessary here, because we must ensure that the recursion `depth` always corresponds to the level of `assume`.

Second, for the similar reason, a worker may run into conflicts while processing the clauses newly fetched from `global_DB`. That is, with the worker's current variable assignments, a newly fetched clause can already be in the state of false or asserting. In which case, the worker backtracks to a level where all its clauses are either true or free.

The parallel implementation we described in this section is nondeterministic and worker-aware, because each worker keeps its own solver state, possibly with different set of watched literals. When a thief resumes a stolen work, it replays the assume to get its solver up-to-date for the stolen work. That means, the results of reply depends on which worker happens to steal and resume a particular branch. As we will see in Section 5, the nondeterminism of the program has some consequences on the execution time.

5. Experimental Results

In this section we present some preliminary experimental results on our implementation of Parallel MiniSat. We ran all the experiments on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 Gbytes of memory. Each core on a chip has a 64-Kbyte private L1-data-cache and a 512-Kbyte private L2-cache, but all cores on a chip share a 2-Mbyte L3-cache.

Our implementation runs on the publicly available Cilk-5 distribution (version 5.4.6) without modification.[2] The Cilk source code is first compiled using the source-to-source translator that comes with the Cilk-5 distribution, and the translated C source is then compiled with gcc-4.3 with -O2 flag. The original MiniSat implementation is as well compiled with gcc-4.3 with -O2 flag.

We evaluated our implementation with eight inputs, four of which are satisfiable, and four of which are unsatisfiable. The inputs are chosen based on their sizes (*i.e.*, the number of variables and clauses in the input constraints) and how long our Parallel MiniSat takes to solve them when running on single processor. For the satisfiable inputs, the input size ranges from 300 variables / 1260 clauses to 26986 variables / 132388 clauses, and the execution time on single processor ranges from 13 to 21 seconds. For the unsatisfiable input, the input size ranges from 132 variables / 1527 clauses to 156980 variables / 696581 clauses, and the execution time on single processor ranges from 8 to 21 seconds. For all inputs, we collected the execution time and number of inspects (*i.e.*, the number of clauses examined during propagation) per second, which serves as an approximation of the throughput of the solver. Each data point is the mean of five runs.

Figure 4 shows the execution time of all inputs relative to the execution time running on single processor, with the left graph shows execution time for four satisfiable inputs, and the right graph shows that for the unsatisfiable inputs. In some cases, we are getting significant slow down instead of speedup when executing on multiple processors. We still need further investigation to understand why this is the case. In general, we don't observe any consistent pattern in execution time across the inputs as the number of processors increases. The only consistency we observe across the inputs is that the execution running on single processor is comparable to the execution time of the original MiniSat, despite the spawn overhead.

The execution time, however, is not a good indicator for how the well the parallelization worked due to two reasons. First, our implementation of Parallel MiniSat performs speculative computation, and therefore the amount of work performed varies depending on the number of processors used during execution. Second, the execution time is partially dictated by how much the solver can prune the search space, which heavily depends on the variable ordering heuristics and the effectiveness of the learned clauses. The outcome of using these techniques correlates to what search path the solver explored. The fact that our solver implementation is nondeterministic and worker-aware makes the outcome of using these techniques even more unpredictable. As a result, the amount of work performed from run to run may not be the same even when running with the same number of processors. As we can observe from the results, the standard deviations (not shown in graphs) for the data points when running on multiple processors range from 21% to 177% for the satisfiable inputs and range from 6% to 38% for the unsatisfiable inputs.

For a more informative evaluation of how well the parallelization worked, we must account for the amount of work performed during execution. Thus, we also measured the number of inspects, which is the number of clauses examined during propagation throughout execution, to approximate for the amount of work performed, since the solver spends considerable amount of time doing propagation.

Figure 5 shows the number of inspects per second relative to the number of inspects per second running on single processor, with the left graph showing the satisfiable inputs and the right graph showing the unsatisfiable inputs. One may think of the number of inspects per second as the throughput of the solver. Ideally, one would expect to see a linear growth in throughput as the number of processors increases, if the number of inspects were an accurate accounting for the amount of work performed and if there were no parallel overhead. While these assumptions don't hold in practice, one would like to see at least a steady increase in throughput as the number of processors increases. In our implementation, some inputs show increase in throughput, such as the first and third satisfiable inputs and the second unsatisfiable input, although the increase is not linear; on other inputs, the throughput shows small, zero, and sometimes negative increase when the number of processors increases.

Figure 6 presents the ***unit throughput***, which is the number of inspects per second per core relative to the unit throughput when running on single processor. This figure presents the same information as Figure 5, but is simply graphed differently. Ideally, one would like to see roughly the same unit throughput as the number of

[2] The Cilk-5 distribution can be obtained at `http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz`.

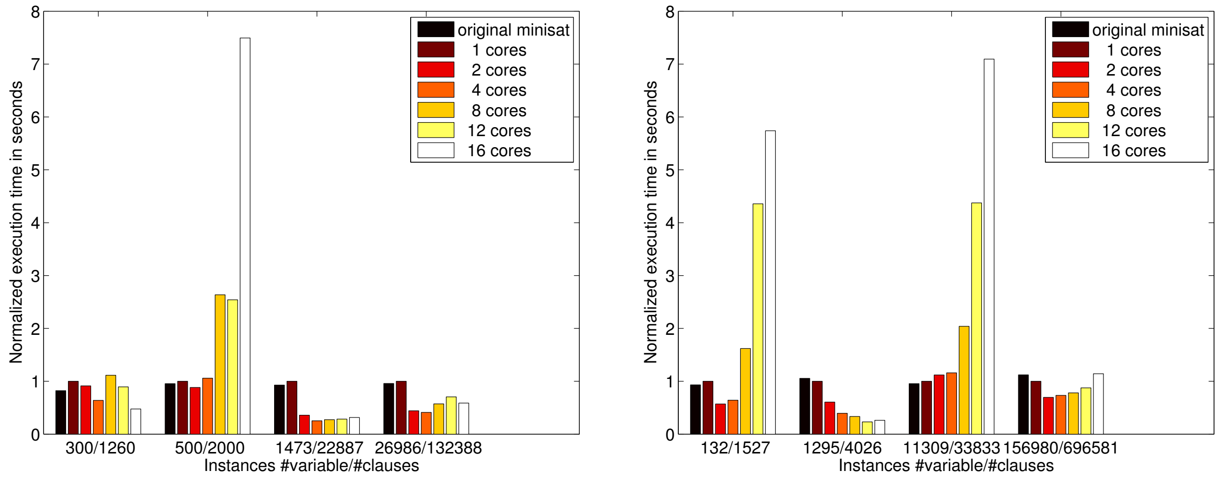

Figure 4: The execution time of the inputs relative to the time running on single processor. On the left, we have the graph for the satisfiable inputs, and on the right we have that of the unsatisfiable inputs. Each set of bars represents one input, and each bar within the set represents the normalized execution time for the original MiniSat and the Parallel MiniSat running on 1, 2, 4, 8, 12, and 16 processors. The X-axis shows the size of each input. The Y-axis shows the execution time normalized by the time of Parallel MiniSat running on single processor.

processors increases. While the throughput of our Parallel MiniSat running on one processor is comparable to the original MiniSat, we see a steady decrease in unit throughput in our Parallel MiniSat as the number of processors increases.

We have few speculation as to why this is the case. First, when running on multiple processors, there are additional overhead such as replaying `assume` and fetching / processing clauses from `global_DB`; both incur additional work and possibly synchronization overhead. Second, the memory usage does grow linearly as the number of processors increases, because each worker keeps its own solver state. In particular, the `clause_DB` can take up quite some space, although this reason alone is not convincing, because in the case such as the first unsatisfiable input, the memory usage is much less than say, the last unsatisfiable input, but the unit throughput is worse. Finally, we wonder if the heuristic of variable ordering doesn't work as well in the parallel setting. In our current implementation, each worker keeps track of its own activity and ordering arrays. The value of a variable's activity depends on how often it is involved in a conflict, and the value gets rescaled from time to time when the activity gets too high. A worker only recalculates the activity of a variable when the variable is involved in a conflict that the worker encounters, so a worker's activity array does not reflect or account for the conflicts that other workers encounter. Depending on which branch a worker ended up stealing, its activity array gets shaped differently. We are wondering whether the variable ordering is less effective in the parallel setting, since each worker's ordering reflects only a fraction of the conflicts that the solver encountered as a whole. All these are speculations, however. We need further investigation in order to draw any conclusion.

6. Conclusion

In this section, we conclude with topics for future investigation. As the results indicated in Section 5, the performance of our implementation unfortunately does not scale as the number of processors increases. This can be caused by the replay overhead, the synchronization overhead when accessing `global DB`, memory bandwidth, or even possibly due to the effectiveness of the ordering heuristic in the parallel implementation. Figuring out how we can improve the performance is perhaps the most immediate task.

As we mentioned in Section 4, our implementation is nondeterministic and worker-aware. Which search path gets explored next highly depends on which worker ends up stealing the branch, since every worker keeps its own solver state, given a particular prefix of `assume`, every worker may propagate differently and use different variable ordering to select the next variable. It is an interesting topic as to how we can make the solver more deterministic.

The most naive deterministic implementation would be to make a snapshot over the solver state before every branch; when a thief steals a particular branch, the thief continues the search with the snapshot of the solver state taken before the first `spawn`. Similarly, if a thief picks up the continuation after `sync`, it continues the search with the snapshot of the solver state taken before the first `spawn`. This implementation is clearly not ideal; the coping overhead aside, it does not take any advantage of the variable ordering or clause learning that one would get from keeping the state after exploring certain search subspace.

If one intends to keep the effect of variable ordering and clause learning, and accumulate knowledge across workers as they explore different subspaces, it seems challenging to make the solver deterministic. In our current implementation, in order to accumulate the effect of clause learning, we had a centralized global clause database that each worker periodically synchronizes to. Furthermore, since we reuse the mechanism of how the original MiniSat keeps track of a clause's watched literals, we ended up having every worker keeping track of its own local database. The nondeterminism is resulted from the fact that workers synchronize to the global database asynchronously, and every worker may select different watched literals for a given clause depending on the variable assignments it made so far. Even if we separate the notion of learned clause database and watched literals so that there is one centralized clause database, and every worker keeps track of its own list of watched literals, the workers would still need to synchronize their lists of watched literals to the clause database whenever a new clause is added, which is likely asynchronous. It is unclear at the moment how to get around these issues and still allow the workers to accumulate the effect of learning.

In terms of the variable ordering, one could conceivably make a snapshot of the order array before `spawn`, and have the thief copy it over upon a steal. With this strategy, however, the question is how we can accumulate the learning effect somehow, such that

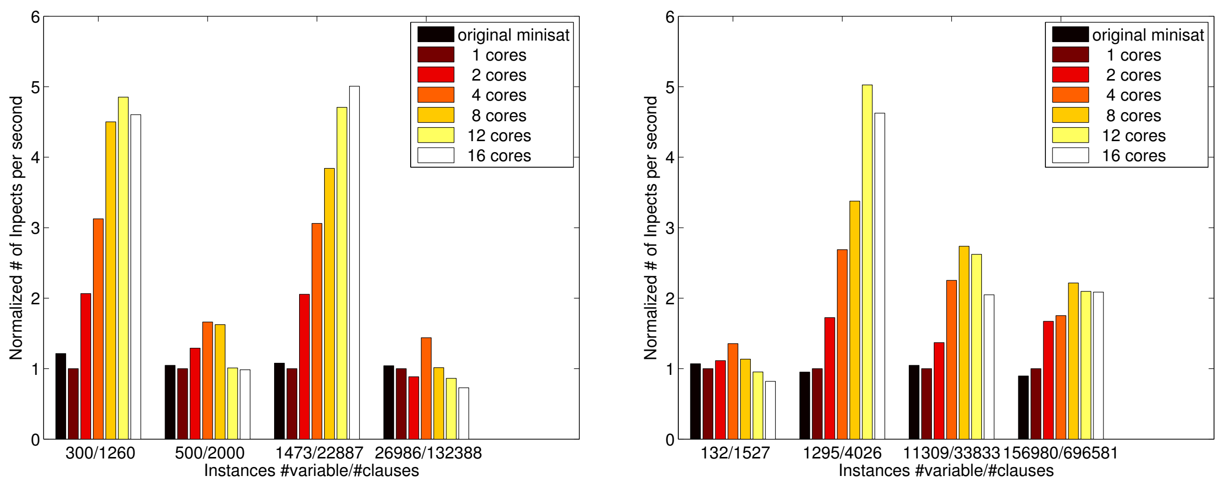

Figure 5: The number of inspects per second of the inputs relative to the number of inspects per second executing on single processor. On the left, we have the graph for the satisfiable inputs, and on the right we have the graph of the unsatisfiable inputs. Each set of bars represents one input, and each bar within the set represents the normalized number of inspects per second for the original MiniSat and the Parallel MiniSat running on 1, 2, 4, 8, 12, and 16 processors. The X-axis shows the size of each input. The Y-axis shows the number of inspects per second normalized by the number of inspects per second of the Parallel MiniSat running on single processor.

the activity reflects the conflicts encountered so far, but not just the conflicts encountered between root of the search tree to this branching point. While these issues seem difficult to get around, and it may be the case that an efficient parallel solver cannot be completely deterministic, we certainly like to move towards that direction as much as possible.

REFERENCES

[1] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT Department of EECS, September 1995.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, Santa Margherita Ligure, Italy, May 2003. Available from `http://minisat.se/Papers.html`.

[4] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128:75–90, 2005.

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[7] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. In *Journal on Satisfiability, Boolean Modeling and Computation*, pages 245–262, 2009.

[8] Franjo Ivani, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[9] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Eighteenth national conference on Artificial intelligence*, pages 674–681, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[10] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *PLDI '89*, pages 81–90, June 1989.

[11] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[12] João P. Marques-Silva and Karem A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[13] João P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 675–680, New York, NY, USA, 2000. ACM.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th Annual Design Automation Conference*, pages 530–535, 2001.

[15] Carsten Sinz, Wolfgang Blochinger, and Wlfgang Küchlin. PaSAT — parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205 – 216, 2001. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

[16] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

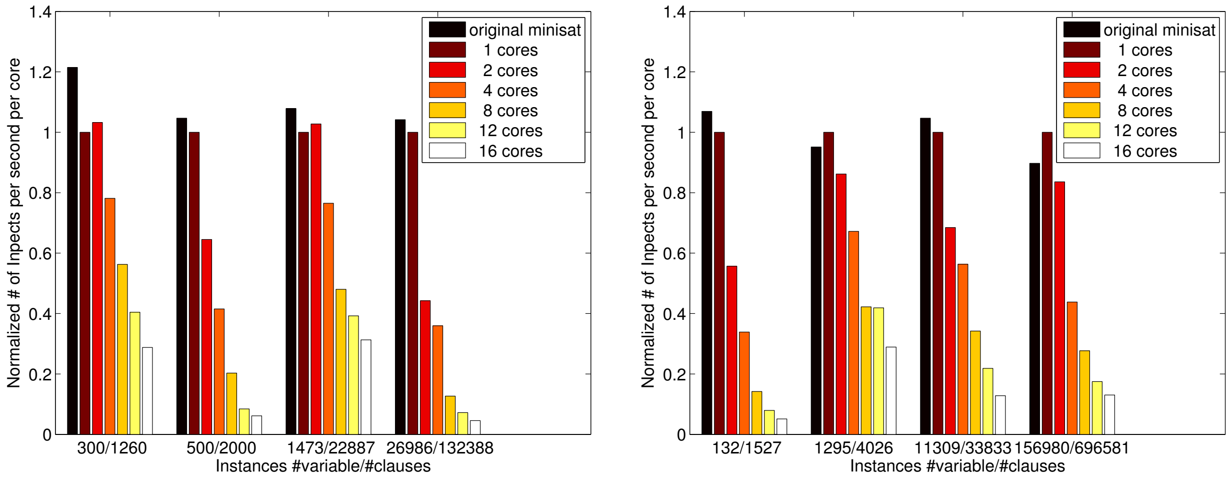

Figure 6: The unit throughput (*i.e.*, the number of inspects per second per core) of the inputs relative to the unit throughput when executing on single processor. On the left, we have the graph for the satisfiable inputs, and on the right we have the graph of the unsatisfiable inputs. Each set of bars represents one input, and each bar within the set represents the normalized unit throughput for the original MiniSat and the Parallel MiniSat running on 1, 2, 4, 8, 12, and 16 processors. The X-axis shows the size of each input. The Y-axis shows the unit throughput normalized by the unit throughput of the Parallel MiniSat running on single processor.