Approaches for Multi-Core Propagation in Clause Learning Satisfiability Solvers

Antti E. J. Hyvärinen[1]* and Christoph M. Wintersteiger[2]

[1] Aalto University, Finland
[2] Microsoft Research, UK
antti.hyvarinen@aalto.fi, cwinter@microsoft.com

Abstract. Parallelization of unit propagation in SAT solvers is a compelling way of obtaining an efficient parallel decision procedure for the propositional satisfiability problem. However, due to the P-completeness of unit propagation, it is challenging to achieve good efficiency in practice. In this article, we present two methods for unit propagation on multi-core systems and their implementation. We throughly evaluate these techniques by comparison to a simulation that estimates a baseline efficiency and by experimental evaluation of an implementation on competition benchmarks. We thereby demonstrate that achieving a speed-up linear in the number of cores is indeed challenging in practice, but also that unit propagation on multi-core systems is feasible in practice.

1 Introduction

Modern SAT Solvers based on Conflict-Driven Clause Learning (CDCL) employ unit propagation (also called Boolean constraint propagation) to determine the consequences of fixing the value of one or more variables in the problem. The Unit Propagation Algorithm computes all implied literals of a partial assignment to the variables in a formula. While computing this unit closure, the algorithm repeatedly iterates through a large number of clauses in the formula, resulting in high memory bandwidth utilization. The solver spends a significant portion, often 90 % and more [6], of its run time propagating units. Fast computation of the closure is therefore of paramount importance to the efficiency of the solver.

Modern computers host increasing numbers of CPUs and cores. Therefore, parallelization of the unit propagation has received increasing interest recently (see, e.g, [8]). However, many parallelization techniques require expensive synchronization which increases the burden on the memory bus, and it is not clear how much speed-up, if any, can be obtained by parallelizing unit propagation. Given that the problem is P-complete [3], it is clear that parallelizing unit propagation efficiently is challenging in theory.

In this article, we study parallelizations of the unit propagation algorithm using a natural parallelization model and a realistic, fast implementation based on the widely used SAT solver MiniSat (version 2.2.0) (see [2]). We show that

* This work was performed while an intern at Microsoft Research.

parallel unit propagation is feasible in practice and that it does deliver speed-ups on competition-type benchmarks. At the same time we show that the challenges set by complexity theory do indeed translate to practical applications and that it becomes increasingly difficult to achieve linear speed-ups as the number of processors grows.

2 Preliminaries

Let x be a Boolean variable and $\neg x$ its negation. Given a set V of Boolean variables, the set $\{x, \neg x \mid x \in V\}$ is the set of *literals*. A *clause* is a disjunction of literals and a *propositional formula* is a conjunction of clauses. When convenient, the clauses are interpreted as sets of literals. No clause contains two literals over same variable. A truth assignment τ is a set of literals such that for no variable x, both x and $\neg x$ are in τ. If the assignment contains a literal over every variable in a formula, it is called complete, otherwise it is called partial. A given formula ϕ is *satisfiable* if there is a truth assignment τ such that each clause of ϕ contains a literal in τ, and *unsatisfiable* otherwise. The propositional satisfiability problem (SAT) is to determine whether a formula ϕ is satisfiable. If l is in τ then l is said to be true and $\neg l$ false, whereas literals l such that τ contains neither l nor $\neg l$, are unassigned.

A *Conflict-Driven Clause Learning (CDCL) SAT solver* determines the satisfiability of a given formula ϕ by searching for a satisfying truth assignment or showing that no such assignment exists. During the search, a CDCL solver derives new clauses that are guaranteed to implied by ϕ. A central part of the CDCL search is to extend a truth assignment candidate τ by *unit propagation*. The extension $\mathrm{UP}(\tau, \phi)$ is the smallest set of literals containing τ such that $UP(\tau, \phi)$ contains every literal for which $(\phi \wedge \tau) \Rightarrow l$. Such literals are called *implied*, and the unit propagation process, effectively determining the implied literals, constitutes the majority of the run time of a SAT solver (cf. e.g. [6]).

3 Sequential Unit Propagation Closure

The unit propagation closure computation employed in modern SAT solvers iteratively identifies implied literals by analyzing a set of clauses against a partial truth assignment. As the number of clauses in a formula can be large. For instance, the benchmark problems used for the application track of the SAT competition in 2009[3] had, on average, just under 900,000 clauses. Most unit propagation algorithms therefore try to minimize the number of clause evaluations while computing the closure.

A typical unit propagation closure algorithm is shown in Fig. 1. The algorithm keeps a *propagation queue UPQ* containing literals that are to be added to the partial assignment τ. Typically, *UPQ* contains a single (assumed) literal when the algorithm is started, and implied literals are appended to the end of

[3] see `http://www.satcompetition.org/`

the queue when they are identified during execution. In every iteration, the algorithm removes a literal from the head of the queue, and determines whether the removed literal results in new implied literals under the current partial assignment τ.

Most modern CDCL solvers (see, e.g, [2, 6, 1] use unit propagation algorithms based on lazy data structures like *watched literal schemes* [11], where some (usually two) literals in each clause are being 'watched'. When a literal p is taken from UPQ, a clause containing $\neg p$ becomes unit, when all but one of the literals in the clause have their negation in τ, i.e., the remaining literal is forced to be assigned to true if the formula is to be satisfied. The watched literal scheme helps to optimize this deduction, because it minimizes both the number of clauses and literals that need to be investigated to determine implied literals. To investigate properties of the algorithm, we divide the clauses into three categories:

a) clauses that contain at least one unassigned literal l which is not watched,
b) clauses where only the watched literals l_1, l_2 are unassigned in τ, and
c) clauses that do not contain any unassigned literals.

Note that case a) is simple unit propagation, case b) detects the implication of l_1 when $\neg l_2$ is assigned and case c) constitutes a *conflict* if all literals in the clause are assigned to false.

Each literal l is associated with a (possibly empty) watch list $WL[l]$ of clauses that have l as one of their watches. Once l is taken from the unit propagation queue, all the clauses in $WL[\neg l]$ are inspected for a new, unassigned literal l_n to be watched. If one can be found, the clause is removed from the current watch list and added to $WL[l_n]$. If no such literal can be found, the clause is unit and the other watched literal is pushed onto the unit propagation queue. The unit propagation closure algorithm terminates once the unit propagation queue is empty or when some implied literal cannot be assigned because its negation has already been assigned.

Many solvers implement variations of the algorithm depicted in Alg. 1 and incorporate further optimizations. For example, in MiniSat, the watch list entries contain one literal from the clause c and a pointer to the clause. If this literal is assigned to true, the clause is not inspected at all, potentially avoiding a superfluous memory lookup. Also, most implementations of the two-watched literal scheme keep the watches as the two first literals of the clause. Thereby, no additional storage is required for remembering which of the literals are watched. Note that, other parts of the CDCL algorithm, e.g., conflict analysis, may require the possibility of recovery of the order in which the literals were assigned, after the unit propagation closure has been computed.

4 Parallel Unit Propagation Closure

A straightforward parallelization of unit propagation is obtained by simply instructing multiple worker threads to obtain literals from the propagation queue, such that every thread propagates a different literal at any time. The downside

Algorithm 1 Unit Propagation Closure using a two-watched literal scheme

```
Vars
   a (non-empty) unit propagation queue $UPQ$;
   an array of watch lists $WL[]$;
   a truth assignment $\tau$
Function propagate():
1  while $UPQ$ is not empty:
2    remove a literal $p$ from $UPQ$
3    for each clause $c$ in $WL[\neg p]$:
4      let $\neg q$ be the other watch of $c$
5      if $c$ contains an literal $l \neq \neg q$ that is unassigned in $\tau$:
6        remove $c$ from $WL[\neg p]$
7        append $c$ to $WL[l]$
8        replace $\neg p$ by $l$ as the new watch of $c$
9      else if $q$ is unassigned:
10       add $\neg q$ to $\tau$
11       append $\neg q$ to $UPQ$
12     else:
13       return conflict
14 return no conflict
```

of this approach is that a significant amount of expensive synchronization is required to avoid rendering the state of the algorithm inconsistent. Manthey [8] therefore proposes to partition the clause database, such that every thread has exclusive access to its partition. It is clear that this partitioning has a significant influence on the speed-up of the parallelization. Some partitioning schemes have been suggested (e.g., [7]), but no general consensus about what constitutes a 'good' partitioning scheme exists.

Here, we explore a variation of the algorithm that does not partition the formula. Instead, we propose to parallelize unit propagation using light-weight synchronization primitives like lock-free data structures [5, 10] and hardware memory barriers [9]. In particular, the unit propagation queue, the truth assignment, each watch list and each clause either need to be guarded by a lock or replaced with lock free data structures. Furthermore, when two watch lists are being accessed simultaneously (lines 3 and 6 in Alg. 1 read from watch lists and line 7 writes to a watch list), a more elaborate locking scheme is required to avoid deadlocks.

This approach is expected to result in good load balancing between the threads as long as a sufficient amount of work is available in the unit propagation queue at all times. Furthermore, the contention is expected to be low in the clauses and the watch lists since it is fairly unlikely that two threads would request access to the same locks due to the high number of clauses and literals in typical application formulas.

Algorithm 2 depicts the top-level strategy of this approach, using the procedures given in Figures 3, 1, and 2. The main loop of the algorithm (Alg. 2) starts N threads and ensures the thread group is able to detect termination, propagate, and avoid deadlocks as mentioned above. Each thread is in one of the states *idle*, *busy* or *conflict*. The states, initialized to *busy* (line 1), are stored in *TH[]*. The parallel part of the algorithm is on lines 2 – 15. Once a thread has finished executing (line 6), the thread waits for other threads on line 15. After each thread reaches line 15, the main thread continues execution and determines the exit value of the unit propagation closure. The value is *conflict* if at least one of the threads found a conflict and *no conflict* otherwise.

Algorithm 2 Parallel Unit Propagation

```
Vars
   id, the current thread identity
   TH[i] states for each thread i
Function parallelPropagate():
1  let TH[i] = busy for each thread i
2  begin parallel:
3     while true:
4        let p = getWork()
5        if p = ⊥:
6           break
7        let TMP = a thread-local empty list
8        let state = litPropagate(p, TMP)
9        lock(TH[])
10       let TH[id] = state
11       release(TH[])
12       restoreWatches(TMP)
15 end parallel
16 if TH[i] = conflict for some thread i:
17    return conflict
18 else return busy
```

The termination of the unit propagation closure algorithm is detected on lines 4–6, and the corresponding function **getWork**() is shown in Alg. 3. The function first checks if one of the threads has found a conflict and returns immediately if this is the case. New work is extracted in the critical sections on lines 17-29, first from the unit propagation queue and then from the inner propagation queue (see below for definition). If no work is found, the state of other threads is checked on lines 32-37 and if all threads are in state *idle*, the final thread sets the variable *cont* to false to indicate that the unit propagation closure is computed. Otherwise, the thread sets its state to *busy* on lines 43-46. Once a thread has set its state, it exits the critical section. If work was found or termination was detected, the respective result is returned on lines 14 or 41. Otherwise the thread repeats the procedure.

Unit propagation for a single literal is handled on lines 7–8 in Alg. 2, and the corresponding function *litPropagate()* is shown in Alg. 1. The function starts by locking the watch list of $\neg p$ on line 1 and inspects all the watched clauses on lines 2–26. Each clause c is locked to avoid two threads changing the same clause simultaneously. The case where c contains an unassigned non-watch is handled on lines 5–12. To avoid the deadlock, an attempt to lock the new watch list is done on line 7. If the watch was not locked, the new clause is updated to the watch list. Otherwise the watch is stored in the thread local list *TMP* on line 11. The watch is then updated on line 12. The case where c does not contain an unassigned non-watch is handled on lines 13–25. If the other watch $\neg q$ is unassigned, it is assigned, implied and added to the unit propagation queue on lines 15 – 20. Otherwise either a conflict is detected or the other watch was already assigned true. In the former case the algorithm returns on line 26. In the latter case the algorithm studies the next clause in the watch list.

The third parallel phase in the unit propagation closure of Alg. 2 is described on lines 9–12. If propagating the literal on line 8 resulted in a conflict this is recorded to the state of the thread on lines 9 – 11. The function *litPropagate()* stored the watches that it could not immediately update to *TMP*. These are restored on line 12 with the *restoreWatches()* function described in Fig. 2. The function *restoreWatches()* takes an entry from the thread local list *TMP* and attempts to place it in the corresponding watch list on lines 2–6. If the function *litPropagate()* has already been called for the literal $\neg l$, then it is possible that the clause c was still in *TMP*. Therefore $\neg l$ is placed on the inner propagation queue, where it will be found by the function *getWork()* later. Finally the function returns once all clauses have been removed from *TMP*.

4.1 Implementing the Parallel Algorithm

To improve overall performance, some optimizations may be made in an implementation of the parallel unit propagation algorithm:

The locking system in the Alg. 2 can be more light-weight as in many cases the contention is expected to be low. Furthermore, many of the operations can be performed with a single, atomic call to *InterlockedCompareExchange()* or the *CompareAndSwap()* instruction provided by most operating systems. In particular, lines 9–11 of Alg. 2 may be replaced by *InterlockedCompareExchange()*; and in Alg. 3, lines 2–6 needs no locking, lines 21–29 need no locking as long as there is a memory barrier on line 30.

The locks for the watch lists, represented by the respective literals (for example, line 1 in Fig. 1), can be implemented as spin locks on the watch data structure. A similar approach can be taken for the locks of clauses.

The operations on the truth assignment can be implemented without locks on lines 14–21 using atomic *InterlockedCompareExchange()*.

Algorithm 3 Termination detection and work retrieval algorithm

```
Literal getWork(UPQ /* unit propagation queue */,
                IPQ /* inner propagation queue */,
                id  /* thread identity */,
                TH[] /* thread states */) {
  Literal r = ⊥.
  while(true) {
    lock(TH[]);
    bool conflict = false;
    if (TH[i] is conflict for some thread $i$)
      conflict = true;
    release(TH[]);

    if (conflict) {
      return ⊥;
    }
    else {
      lock(UPQ);
      if (UPQ non–empty) {
        $r =$ a literal from UPQ;
        remove $r$ from UPQ;
      }
      release(UPQ);
      if ($r = \perp$) {
        lock(IPQ);
        if (IPQ non–empty) {
          $r =$ a literal from IPQ;
          remove $r$ from IPQ;
        }
        release(IPQ);
      }
      if ($r = \perp$) {
        lock(TH[]);
        TH[id] = idle;
        if (TH[i] = idle for all threads $i$) {
          cont = false;
        }
        release(TH[]);
      }

      if (cont = false)
        return ⊥;
      else if ($p \neq \perp$) {
        lock(TH[]);
        TH[id] = busy;
        release(TH[]);
        return r;
      }
    }
```

```
**Vars**
    $WL[l]$, the watch lists for each literal $l$; $UPQ$, the unit propagation queue
**Function** *litPropagate(p, TMP)*:
    *lock($\neg p$)*
    **for** each clause $c$ in $WL[\neg p]$:
        *lock(c)*
        **let** $\neg q$ be the other watch of $c$
        **if** $c$ contains an unassigned literal $l \neq \neg q$:
            remove $c$ from $WL[\neg p]$
            **if** *trylock(l)*:
                append $c$ to $WL[l]$
                *release(l)*
            **else**:
                append $(l, c)$ to $TMP$
            replace $\neg p$ by $l$ as the new watch of $c$
        **else**:
            lock the truth assignment
            **if** $q$ is unassigned:
                *lock(UPQ)*
                assign $\neg q$ to true
                append $\neg q$ to $UPQ$
                *release(UPQ)*
            **else if** $\neg q$ is assigned *false*:
                release the truth assignment
                *release(c)*
                *release($\neg p$)*
                **return** *conflict*
            release the truth assignment
        *release(c)*
    *release($\neg p$)*
    **return** *no conflict*
```

Fig. 1. The Literal Propagation Algorithm

```
**Vars**
    WL[l], the watch lists for each literal l; IPQ, the inner propagation queue;
**Function** restoreWatches(p, TMP):
while TMP is not empty:
    let (l, c) be a pair in TMP
    if trylock(l):
        append c to WL[l]
        remove (l, c) from TMP
        release(l)
        if l has been propagated:
            lock(IPQ)
            append l to IPQ
            release(IPQ)
return
```

Fig. 2. The watch restoration algorithm

Extra work is avoided by checking the clauses c on lines 2–26 in Fig. 1 if the literal p is from the inner propagation queue by storing the index of the last clause that was inspected in each watch list.

5 Experimental Results

To assess the efficacy of our algorithms we evaluate them on instances from the application track of the SAT competition 2009[4]. Our implementation is based on MiniSat (with SatELite preprocessing (simp) and without (core) as indicated). We do not compare to other parallel SAT solving approaches as the goal of this work is to assess the feasibility of parallel unit propagation and not to achieve larger speed-ups than other methods. We would like to note that the speed-ups of our parallelization are far inferior to those achieved, e.g., by portfolio-based parallel SAT solvers (cf. e.g. [4]).

5.1 Simulation Experiments

Before assessing our implementation, we are interested in a worst-case model for unit propagation. We therefore present an evaluation of a simulation based on worst-case assumptions to provide baseline data to compare actual implementations to. The distributions for queue sizes and memory jumps, and the memory sizes are obtained from the benchmark files.

Our simulation is a program which holds a memory of size M which is then accessed by N threads. At each iteration, a number n_q of literals is selected from a propagation queue size distribution. Each thread removes literals from the queue

[4] `http://www.satcompetition.org/`

one at the time until the queue is empty, and makes a non-atomic memory swap operation between the previous location and a location from a memory access difference distribution. The memory accesses of each thread are randomized using different seeds to avoid threads looking at the same locations and hence giving the parallel code unfair speedup by cache locality. This corresponds roughly to a case where each literal in the propagation queue watches a single clause.

Figure 3 shows characteristics for the unit propagation queue size for the instances. The left-hand side graph shows the average and maximum number of propagations in unit propagation closure computation. This number is misleading in this context, as typically all the propagations cannot be done simultaneously since the unit propagation closure computation involves chains of propagations. The right-hand side graph shows a more realistic figure, where the size of the propagation queue is measured at the end of each chain

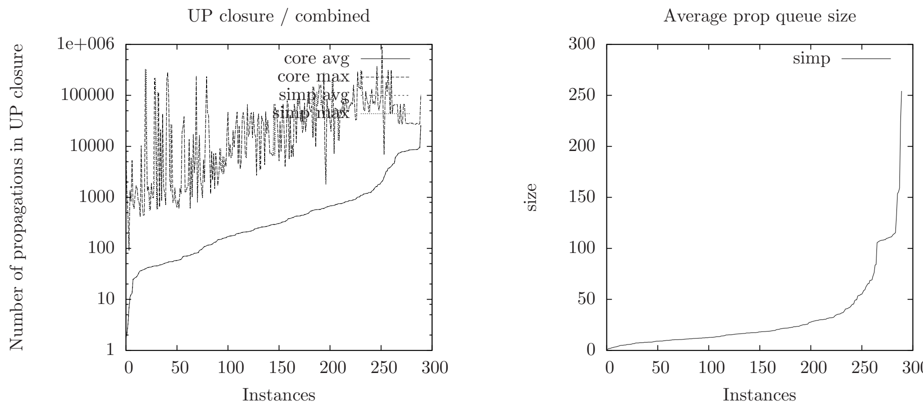

Fig. 3. Unit propagation queue sizes in unit propagation closure (left) and at a given parallelizable snapshot (right)

Figure 4 (right) shows the time used in computing a unit propagation closure. This is the time that the threads can be running simultaneously and hence should be parallilized to obtain speedup.

The results in Fig. 5 are obtained using the previously collected distributions (sample size is 1000 for both memory jumps and queue sizes). Each simulation starts by running a sequential version of the code for 10 minutes and computing the number of performed swaps. Then the parallel code performs the same number of swaps. The sequential time (10 minutes) is then divided by the wall-clock time used by the parallel execution resulting in the reported speedup figures. Based on the figures, it seems that in majority of the cases the parallelization results in significant slowdown of the execution in the worst-case model used in the experiments.

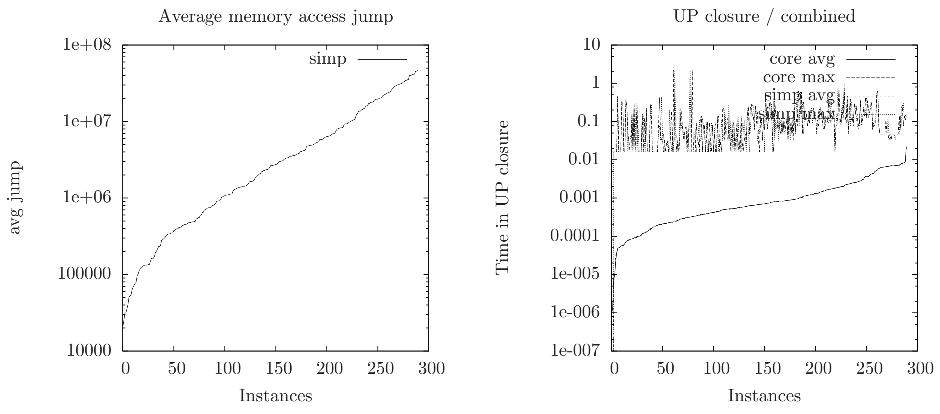

Fig. 4. Average memory access jumps (left) and the average / max time used in unit propagation closure

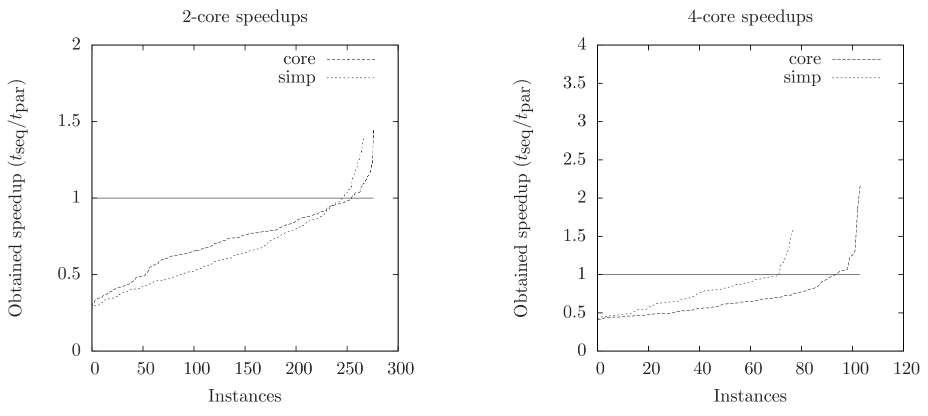

Fig. 5. Simulation speedup for two and four threads

As the number of literals in the propagation queue is the main factor limiting the parallelization in the model, it is interesting to study what results would be obtained if the size of the queue was larger. Figure 6 shows the effet of multiplying the queue size by 1.7.

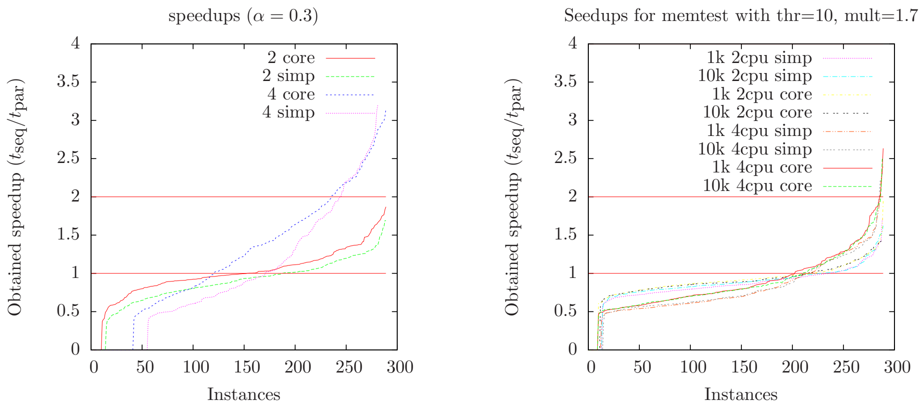

Fig. 6. Simulation speedup for two-decision solver where overlap is 0.3

5.2 Experiments on the Implementation

Figure 7 shows the speedups obtained by running the parallel algorithm described in Sect. 3. The left-hand side graph illustrates the effect of using two cores instead of a single core for executing the same code. In most cases there seems again to be a slowdown, which is roughly in line with the results from the simulations. It would however seem that in the more difficult instances there are some gains and the slowdown is not as significant. As the locking in the code results in slowdown also in sequential case, it is useful to compare the results against the unaltered MiniSat in the right-hand side graph.

6 Conclusions

6.1 Future Work

The operations on the inner propagation queue can be implemented using a lockless queue. The same is true for the unit propagation queue, but then either the atomicity of the assignment and appending the implied literal on lines 17 and 18 in Fig. 1 need to be ensured or the assignment order needs to be stored in the truth assignment so that the order can later be recovered for the conflict analysis in other parts of the CDCL algorithm.

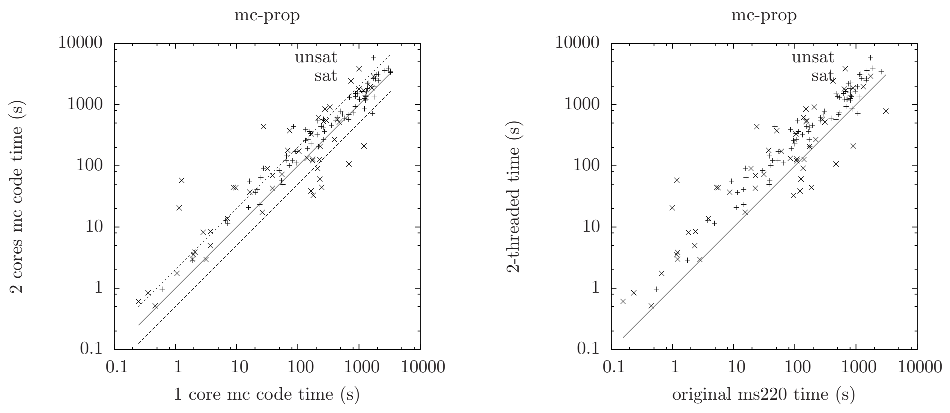

Fig. 7. Speedup obtained with the implementation (two threads)

It feels likely that the clause level synchronization could be avoided in many cases by storing short clauses only in watch lists. At least binary clauses should fit into this nicely.

In the current implementation there is a somewhat rare assertion failure in memory management which has never been observed in sequential code, once in 300 instances in two-threaded version and seven times in four-threaded version. This could either result from memory exhaustion or a race condition in MiniSat internal memory management.

It would be interesting to work more on the simulation model, as clearly the assumption that each watch list contains only a single clause is invalid. Perhaps more insight could be obtained by studying realistic watch list sizes and their effects to the speedups. This could give further insight in how to improve the parallel unit propagation.

References

1. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University, Altenbergenstr. 69, 4040 Linz, Austria (Aug 2010)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT. LNCS, vol. 2919, pp. 502–518. Springer (2004)
3. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: Limits to Parallel Computation: P-Completeness Theory. Oxford University Press (1995)
4. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. JSAT 6(4), 245–262 (2009)
5. Hart, T.E., McKenney, P.E., Brown, A.D.: Making lockless synchronization fast: performance implications of memory reclamation. In: International Parallel and Distributed Processing Symposium. IEEE (2006)
6. Hölldobler, S., Manthey, N., Saptawijaya, A.: Improving resource-unaware SAT solvers. In: Proc. LPAR. LNCS, vol. 6397, pp. 519 – 534. Springer (2010)

7. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Proc. LPAR (Yogyakarta). LNCS, vol. 6397, pp. 372–386. Springer (2010)
8. Manthey, N.: Parallel SAT solving – using more cores. In: Easychair Proc. Workshop on Pragmatics of SAT. pp. 1–14 (2011)
9. McKenney, P.E.: Memory barriers: a hardware view for software hackers (Jun 2010), available on-line at `http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf`
10. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Symposium on Principles of Distributed Computing. pp. 267 – 275. ACM (1996)
11. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC. pp. 530–535. ACM (2001)