Parallel SAT Solver

Summary

We implemented a non-deterministic parallel k-SAT solver using data parallelism and OpenMP. We were able to quickly solve the hardest SAT-competition [2] level problems (under a few seconds on the GHC cluster).

Background

Boolean satisfiability is a classic computer science problem which requires finding an assignment to variables which satisfy a particular boolean expression. The classical formulation expresses a boolean expression in Conjunction Normal Form (CNF). A CNF is a conjunction of multiple clauses, where each clause is a disjunction of literals, and each literal is either a variable or the negation of a variable. An example of a CNF is e.g.

$F = C1 \wedge C2 \wedge C3$,
$Ci = l1 \vee l2 \vee l3$,
li = xi or !xi

Applications of SAT solvers include circuit board design [9], cryptography, and graph problems or even puzzles like Sudoku.

Some SAT solvers can only find the solution for satisfiable problems, whereas other solvers, known as complete solvers, can operate on satisfiable or unsatisfiable problems. The former class of solvers uses fast data-structures and reasoning techniques on partial assignments to solve problems. Incomplete Sat solvers, mostly based on local search, mainly perform modifications on a full assignment using ”randomized” flipping decisions. In general, these solvers are less complex. Incomplete solvers are very strong on satisfiable random benchmarks.

Other student groups have worked on complete parallel SAT solvers in previous versions of this course [6, 7, 8], including using message passing, threads and CUDA, which some of which we used in our project as well. However, the previous groups did not use data parallelism or SIMD, and this is the novel aspect of our project.

The main idea here is to use multi-bit boolean assignments (stored in ints of different width) in order to compute multiple combinations at the same time. To see why this is useful, consider the 2-bit assignments on the formula "x and y". There are 16 possible 2-bit assignments of which 7 evaluate to a non-zero value. Drawing multi-bit assignments randomly, the probability of hitting a non-zero multi-bit value is 7/16 , while in the conventional boolean situation this probability is 1/4. In general, the probability is 1 − (3/4)^p using the p-bit assignments. The above example shows that probability to hit a solution using random sampling

MBA’s increases using more bits. In case multi-bit assignments can be used in approximately the same computational time as booleans, solutions can be found faster.

By representing booleans with single bits, the problem is approachable with data parallelism by using ints to represent blocks of booleans. Our key data structure is the *uint{k}_t*, where k=8,16,32,64. This is our "SIMD vector". For different auxiliary functionality, we’ve used a blocking queue, a set, and a queue. Most computation is boolean operations and assignments, &, |, !. There are also set and queue operations.

Approach

The Algorithm

The solver is an incomplete SAT solver focused on local search. In contrast to complete SAT solvers, they are less complicated and work with full assignments. The generic structure of local search SAT solvers is as follows: An assignment ϕ is generated, earmarking a random boolean value to all variables. By flipping the truth values of variables, ϕ can be modified to satisfy as many clauses as possible of the formula at hand. If after a multitude of flips ϕ still does not satisfy the formula, a new random assignment is generated. In the case of our solver, variables are flipped using a technique called "unit propagation" [3].

To understand how unit propagation works, let's define unit clauses as clauses that are composed of a single literal, in conjunctive normal form. Because each clause needs to be satisfied, we know that this single literal must be true. If a set of clauses contains the unit clause literal l, the other clauses are simplified by the application of the two following rules:

1. every clause (other than the unit clause itself) containing l is removed (the clause is satisfied if l is)
2. in every clause that contains !l (not l) this literal is deleted (!l can not contribute to it being satisfied).

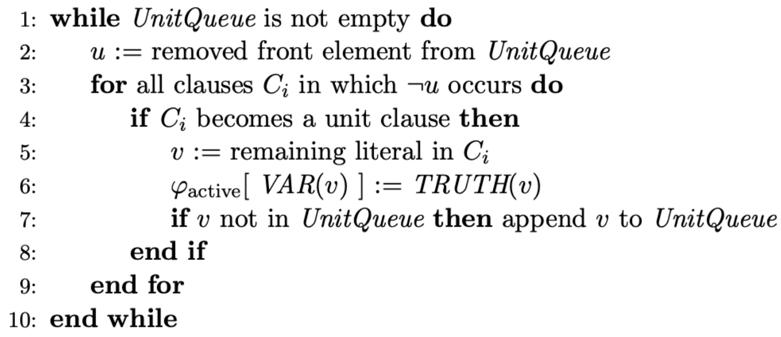
The application of these two rules lead to a new set of clauses that is equivalent to the old one. For example, the following set of clauses can be simplified by unit propagation because it contains the unit clause a.

{a or b, !a or c, !c or d, a}

Since a or b contains the literal a, this clause can be removed altogether. Since !a or c contains the negation of the literal in the unit clause, this literal can be removed from the clause. The unit clause a is not removed; this would make the resulting set not equivalent to the original one. The effect of unit propagation can be summarized as follows.

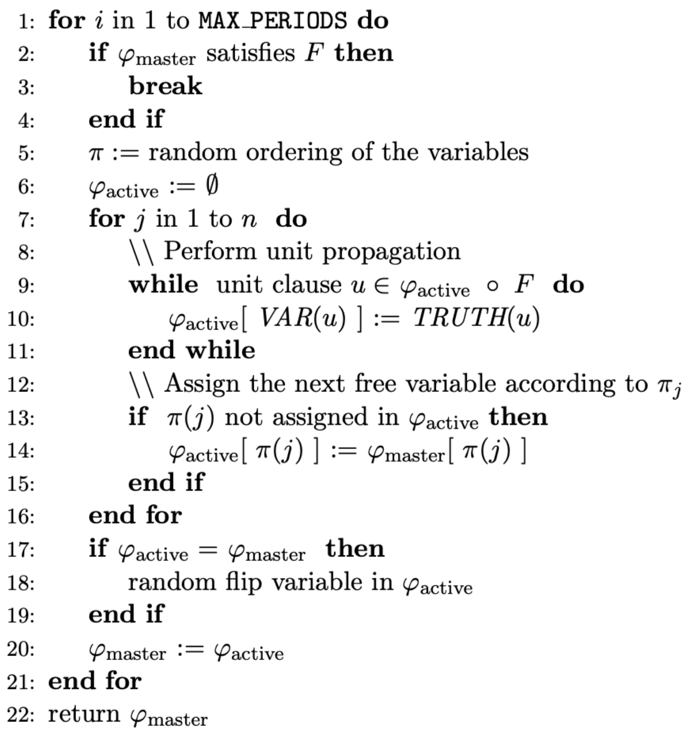
{a or b,	!a or c,	!c or d,	a}
{removed,	!a deleted,	unchanged,	unchanged}
{	c,	!c or d,	a}

In the SAT solver, unit propagation was implemented using a queue (for the sequential unit propagation at least):



```
while $UnitQueue$ is not empty do
    $u$ := removed front element from $UnitQueue$
    for all clauses $C_i$ in which $\neg u$ occurs do
        if $C_i$ becomes a unit clause then
            $v$ := remaining literal in $C_i$
            $\varphi_{\text{active}}[\ VAR(v)\ ] := TRUTH(v)$
            if $v$ not in $UnitQueue$ then append $v$ to $UnitQueue$
        end if
    end for
end while
```

Our solver flips variables in so-called periods. Each period starts with an initial assignment (referred to as master assignment ϕ_master), an empty assignment ϕ_active and a random ordering of all the variables π. First, unit propagation is executed on the empty assignment. Second, the first unassigned variable in π is assigned to its value in ϕ_master, followed by unit propagation of this value. A period ends when all variables are assigned a value in ϕ_active. A new period starts with the resulting ϕ_active as initial ϕ_master and a new ordering of the variables. This is illustrated by the pseudocode below where F designated the SAT formula to satisfy, VAR(u) designated the variable corresponding to literal u, TRUTH(u) corresponds to the value that the remaining literal u in the unit clause must have in order to satisfy the clause.



```
for $i$ in 1 to MAX_PERIODS do
    if $\varphi_{\text{master}}$ satisfies $F$ then
        break
    end if
    $\pi :=$ random ordering of the variables
    $\varphi_{\text{active}} := \emptyset$
    for $j$ in 1 to $n$ do
        \\ Perform unit propagation
        while unit clause $u \in \varphi_{\text{active}} \circ F$ do
            $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
        end while
        \\ Assign the next free variable according to $\pi_j$
        if $\pi(j)$ not assigned in $\varphi_{\text{active}}$ then
            $\varphi_{\text{active}}[\ \pi(j)\ ] := \varphi_{\text{master}}[\ \pi(j)\ ]$
        end if
    end for
    if $\varphi_{\text{active}} = \varphi_{\text{master}}$ then
        random flip variable in $\varphi_{\text{active}}$
    end if
    $\varphi_{\text{master}} := \varphi_{\text{active}}$
end for
return $\varphi_{\text{master}}$
```

Single-bit SAT Solver

Since the goal of our project was to implement a data-parallel SAT solver, we wanted to start by implementing a reference SAT solver that does not benefit from data parallelism, nor of any other kind of parallelism. This SAT solver served three purposes

- Serve as the baseline for performance. The single-bit SAT solver is **not** a special case of the multi-bit solver with unit data parallelism. It was implemented in the most efficient way possible without any considerations to data parallelism, and its logic works best without data parallelism. The upcoming multi-bit solver was supposed to beat the performance of the single-bit solver.

- Serve as the baseline for correctness. The single-bit solver is easier to implement than the multi-bit solver. In case of a bug in the multi-bit solver, the single-bit solver would provide us with the correct intermediate steps, thus making the debugging experience of the multi-bit solver much easier.

- Help us understand the algorithm. The algorithm was mostly based on [1]. It is not trivial to get right, even when disregarding any form of parallelism. It served as an intermediate milestone between the empty code file and the full-blown parallel multi-bit solver.

After finishing the implementation in C++, we were surprised to see how efficient the single bit solver was without even implementing any parallelism strategy. However, we are going to see now how we were able to extract parallelism out of this algorithm, ultimately implementing the multi-bit solver.

Multi-bit Assignments

The most powerful feature of the multi-bit solver are the multi-bit assignments. The solver explores the usefulness of assigning bit vectors {0,1}^p instead of boolean values to the variables. We will refer to these bit vectors as multi-bit values. A non-zero multibit value refers to a bit vector containing at least one 1. An assignment which assigns multi-bit values to the variables is called a multi-bit assignment (in short MBA). Essentially, the boolean operations are converted to bitwise operations (or becomes |, and becomes &, not becomes ~). Let's consider the following example with 3-bit values.

Let F be the formula "x and !y and (!x or !z)"
By assigning x := 101, y := 001 and z := 111, we calculate F:
101 and !001 and (!101 or !111) = 000

By assigning x := 101, y := 001 and z := 011 however, F evaluates to the value 100. All non-zero multi-bit values verify that the given formula is satisfiable. The last step is to take the least significant bit (can be done with (F & ~(F - 1))) of the final result 100. The LSB will give us the bit position of the valid assignment. In this case, an assignment satisfying F is given by (x,y,z) = (1,0,0).

The idea is that we are now able to test a lot (up to 512 if using SIMD or vector instructions) of combinations at once, effectively reducing the number of periods needed by the algorithm.

A few challenges remain to be solved when shifting from single bit to multi-bit. For example, it becomes harder to find unit clauses and their remaining literals in time less than O(k^2) or even O(k^2 * w) where k is the number of literals in a clause and w is the bit width of the multi-bit values. With multi-bit assignments, a clause can become a unit clause on one or more bit positions. Similarly, remaining literals need to be propagated on one or more bit positions. Here is the idea behind the O(k) algorithm that we used to find the unit clauses using multi-bit assignments.

First, note that if a variable is assigned a boolean value, all clauses in which it occurs with complementary polarity are potential unit clauses. Recall that in the 1-bit situation, a potential unit clause can only be unit on a single literal, while in a multi-bit implementation it can become unit on multiple literals (each on a different bit position). In order to detect on which bit positions a clause is a unit clause we split the computation into two steps:

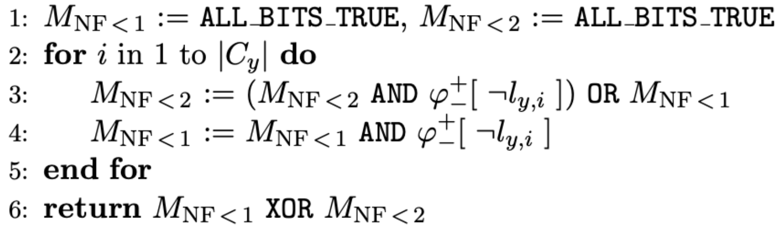
1. Compute the unit mask of a clause - a multi-bit value which is true on all positions with exactly one not falsified literal (denoted by M_nf = 1) and false elsewhere

2. Use the unit mask to quickly determine the newly created unit clauses: All literals that are unassigned at a true position in the unit mask became unit

To compute M_nf = 1, we use two auxiliary masks, M_nf < 1 and M_nf < 2. The masks denote
multi-bit values which are 1 on all positions with less than one (and two, respectively) not falsified literals and 0 elsewhere. Notice that M_nf = 1 := M_nf < 1 XOR M_nf < 2. For each Literal l_i in a clause we update M_nf < 1 and M_nf < 2 by the following two rules:

- M_nf < 2 := (M_nf < 2 and !l_i) or M_nf < 1
- M_nf < 1 := M_nf < 1 and !l_i

$M_{nf} < 2 := M_{nf} < 2 \;\&\&\; ! I_i$



```
$M_{\mathrm{NF}<1} := \texttt{ALL\_BITS\_TRUE}$, $M_{\mathrm{NF}<2} := \texttt{ALL\_BITS\_TRUE}$
for $i$ in 1 to $|C_y|$ do
    $M_{\mathrm{NF}<2} := (M_{\mathrm{NF}<2}\ \texttt{AND}\ \varphi^{+}_{-}[\ \neg l_{y,i}\ ])\ \texttt{OR}\ M_{\mathrm{NF}<1}$
    $M_{\mathrm{NF}<1} := M_{\mathrm{NF}<1}\ \texttt{AND}\ \varphi^{+}_{-}[\ \neg l_{y,i}\ ]$
end for
return $M_{\mathrm{NF}<1}\ \texttt{XOR}\ M_{\mathrm{NF}<2}$
```

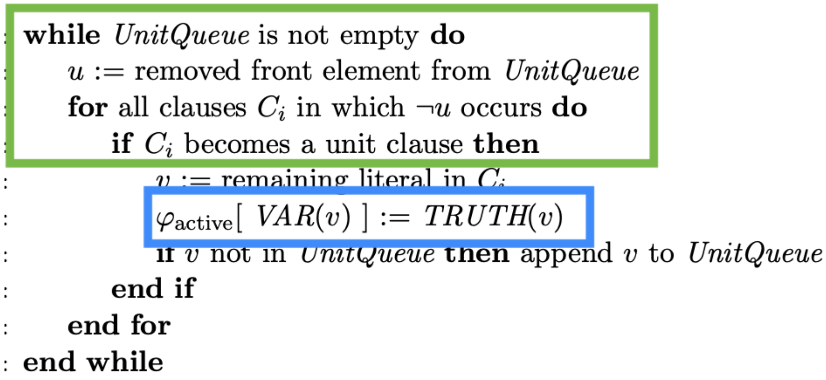
Once we have the mask, we can - for each variable - compute on which bit levels it becomes a remaining literal for the potential unit clause. We simply do x_i = M_nf = 1 and unassigned(x_i), where unassigned(x) is true on the bit positions where x_i is not assigned.

Parallel Unit Propagation

After analyzing the performance of our multi-bit solver, we noticed two things:

- In most test cases, 97-98% of the time is spent inside the unit_propagation() function, trying to find unit clauses.
- We have not used any native thread resources yet, and the processor on the GHC machines has 8 cores.

Both observations made us want to extract task parallelism from the unit propagation method. In particular, we noticed the following opportunity for parallelism in the pseudocode of the algorithm. Recall that the pseudocode of unit_propagation() is given by



```
while $UnitQueue$ is not empty do
    $u$ := removed front element from $UnitQueue$
    for all clauses $C_i$ in which $\neg u$ occurs do
        if $C_i$ becomes a unit clause then
            $v$ := remaining literal in $C_i$
            $\varphi_{\text{active}}[\ VAR(v)\ ] := TRUTH(v)$
            if $v$ not in $UnitQueue$ then append $v$ to $UnitQueue$
        end if
    end for
end while
```

By looking more closely at it, we noticed that there is a read-write conflict between the if condition (the one where we test if a clause is a unit clause) (the green square) and the line where phi_active is assigned (the blue square). In order to determine whether a clause is a unit clause, phi_active must be read.

To parallelize this for-loop, we first considered using atomic read&write. This is the most direct way to parallelize the loop across threads, but we found that atomic operations were extremely slow. We implemented both OpenMP atomic operations and GCC native atomic operations. The biggest reason for this slowness is false sharing.

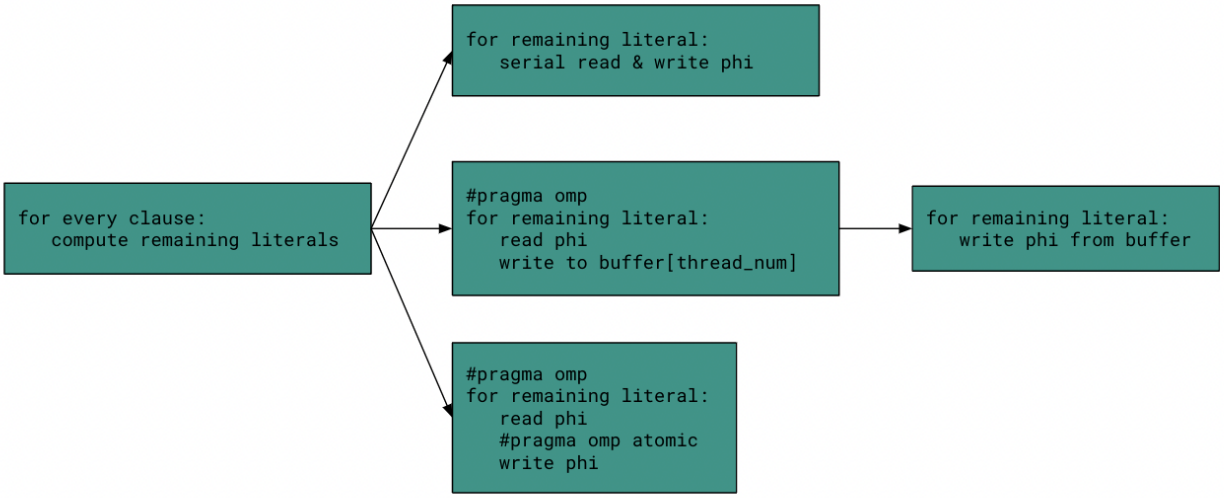
The key contention problem with naive parallelism is that one thread cannot write to the phi_active array while another thread is reading from it. However, we were able to enable parallelism by separating the reading phase and the writing phases. The key data structure is a separate buffer for each thread. We enable parallelism and avoid false sharing as follows

1. In the first step, all the unit clauses and their remaining literals are extracted (the logic from the green square).
2. In the second step, the remaining literals are assigned their corresponding truth value in phi_active (the logic from the blue square)

We chose to parallelize the first step, as most of the time is spent there. We chose not to parallelize the second step because we had already found that the concurrent writes would have been slowed down by false sharing. The most natural framework to achieve this type of fork-join parallelism is OpenMP. Other frameworks were considered but discarded for the following reasons:

- The Cilk implementation of Fork-Join would have worked but is less adapted for this kind of code. It is mainly fitted for a small number (~2) of threads and recursive functions.
- Raw native threads could have been spawned and joined but we did not want to reinvent the wheel by reimplementing OpenMP. OpenMP was already providing the features we needed off the shelf.

The way we used OpenMP to parallelize the work is by splitting all the remaining literals of the current loop iterations into consecutive chunks. Each thread considers its own chunk and generates the next generation of remaining literals determined by that chunk. The implicit barrier between for loops is very useful here as it allows us to clearly separate iterations and assure correctness. The OpenMP "schedule(dynamic)" schedule makes sure that no thread is left idle for too long a time. The algorithm stops when there are no more remaining literals (and thus no more unit clauses) that are being generated. This flowchart represents the ways we looked at parallelizing the operation loop



```
for every clause:
   compute remaining literals
```

```
for remaining literal:
   serial read & write phi
```

```
#pragma omp
for remaining literal:
   read phi
   write to buffer[thread_num]
```

```
for remaining literal:
   write phi from buffer
```

```
#pragma omp
for remaining literal:
   read phi
   #pragma omp atomic
   write phi
```

Parallel Duplicate Removal

Another problem that needs to be addressed are duplicate assignments for different bit positions in the variables. For a given assignment phi, the j-th bit position is called a duplicate if there exists a i < j such that all variables are assigned to the same truth value at bit position i and j. On most benchmarks, duplicates were observed. In some cases even all bit positions became duplicate. Due to the construction of the algorithm, once a bit position is a duplicate, it will remain a duplicate if no intervention is made. Because duplicates reduce the parallel behavior of the algorithm (the whole point of the multi-bit algorithm is to try **different** bit combinations at the same time), we decided to detect duplicates and replace them with a new random assignment.

In order to compute which bit positions are duplicates, we took inspiration from the paper [1]. The solution involved computing a duplicate mask in linear time using assignment matrices and a so-called Hadamard product. The details are left out because they're not too important and the reader can learn more about it by reading section 5.1 of the paper [1]. The final pseudocode looks like this:

```
$m_{\text{duplicates}} := [0]^p$
for $j$ in 1 to $p - 1$ do
    $m_{\text{column}} := [0]^j[1]^{p-j}$
    for $x_i \in \mathcal{F}$ do
        if $x_i$ is assigned to true on the $j$-th bit-position in $\varphi$ then
            $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[x_i]$
        else
            $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[\neg x_i]$
        end if
        if $m_{\text{column}} = [0]^p$ then
            break
        end if
    end for
    $m_{\text{duplicates}} := m_{\text{duplicates}}$ OR $m_{\text{column}}$
end for
return $m_{\text{duplicates}}$
```

Once the mask is known, all the corresponding bit positions can be replaced with random bit positions. The way we parallelized it is by spawning something similar to a daemon thread, that is spawned at the beginning, before the algorithm even starts. This daemon thread keeps popping assignments from a blocking, concurrent queue (the in_queue) that we implemented. Once it has an assignment, it computes a duplicate mask and pushes it to a second concurrent queue, the "out_queue". Every 5 periods, the main thread tries to pop from the out_queue to see if some non-zero duplicate mask could be computed. If that's the case, the main thread replaces all the relevant duplicate bits with random bits. At the end of the algorithm, a "poison pill" is sent to the daemon thread, which exits and is joined by the main thread.

We implemented our blocking queue with two basic concurrency primitives: mutexes and condition variables. The idea is that when pushing, all the threads waiting to pop are notified by the condition variable. When popping, the threads go to sleep as they wait for the condition variable to wake them up. When this happens, they try to pop and return, or otherwise go back to sleep.

The reason why we chose this kind of concurrent queue over a more fine-grained / lock-free queue is that the contention is very small. Every 5 periods or so, there is minimal contention between only two threads. It would not have made sense to implement an ultra complicated queue if the algorithm is barely using it. The essence of the idea is not the queue, but rather the fact that the duplicate removal works in the background.

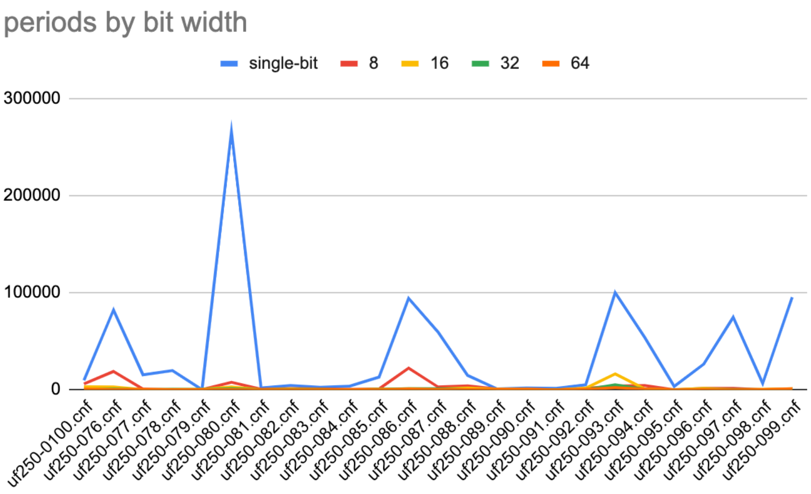
Results

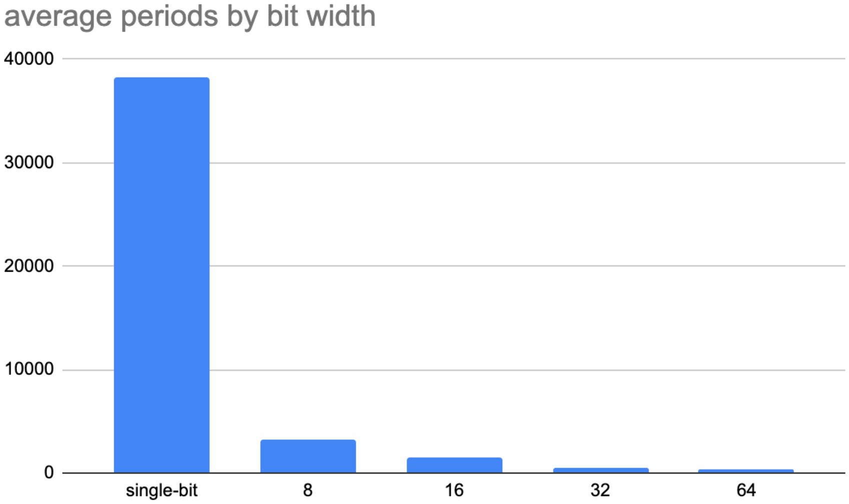
To test the effectiveness of our method, and compare different versions of parallelism, we setup a test harness including build options and test case problems. For problems to test our method, we implemented a version of a random CNF generator using a method called triangle prevention [4]. The generator created problems that were too easy, so we instead imported problems used in a previous SAT solver competition [5]. The problem of generating good tests turned into a simple parsing problem. The random generator that we used is initialized with a constant value in order to keep the same sequence of randomness across different runs.

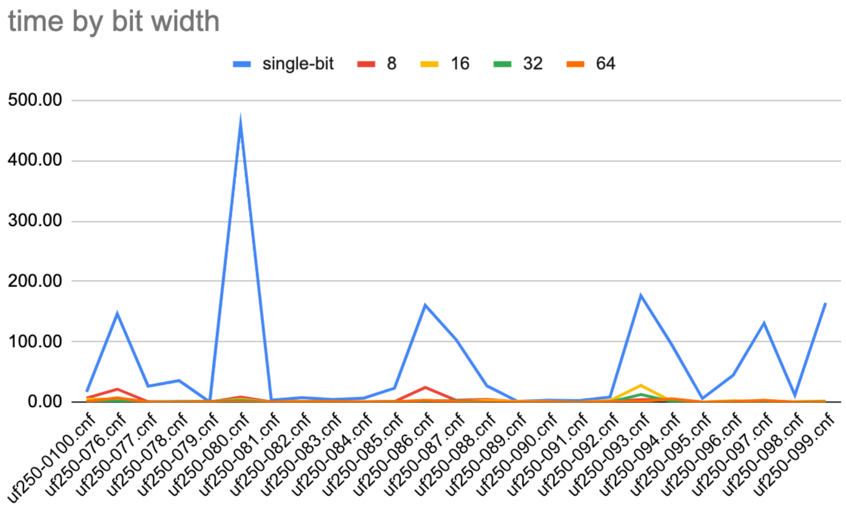
It is not without delight that we realized that even the SAT competition tests were almost instantaneously solved by our algorithm. Only a select few (those with at least 250 variables and 1000+ clauses) were able to stand up to it.

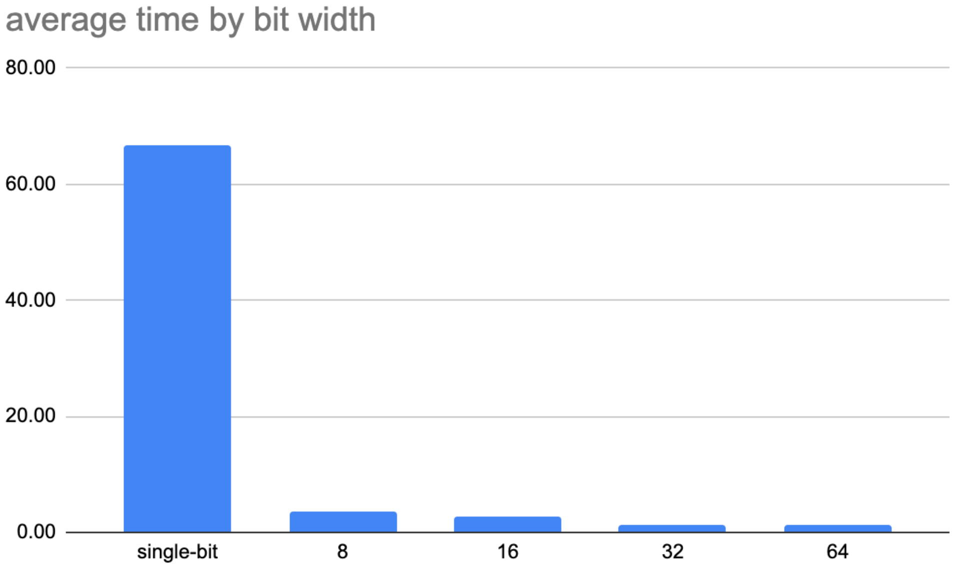
Single-bit vs Multi-bit SAT Solver

As we expected, the multi-bit assignment offers almost linear speedup with respect to the width of the bit vectors. We measured the number of periods needed, the total amount of time as well as the time per period. Those are the results we got after running multiple visions of the solver on 25 of the hardest test cases:

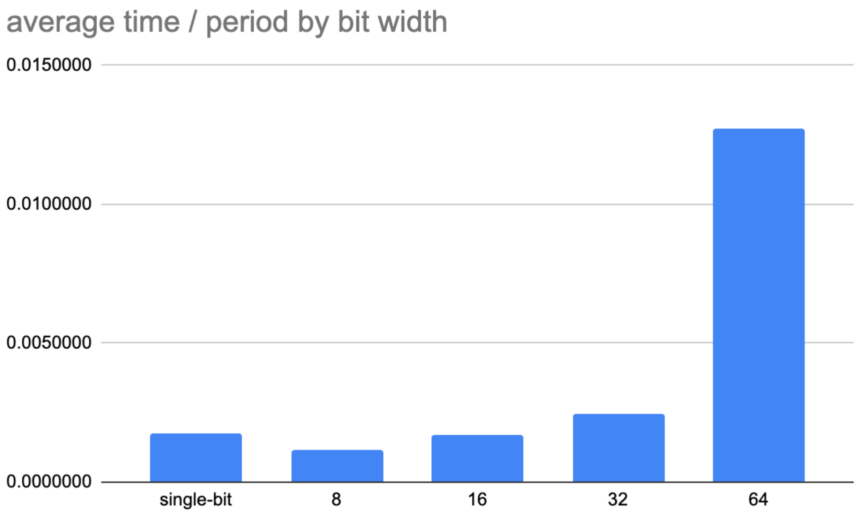








After looking at these charts more carefully, something caught our attention. While the number of periods decreases linearly with the width of the bit vectors, the time remains about the same from 32 bits to 64 bits. We decided to measure the time per period for different bit width and got the following results:

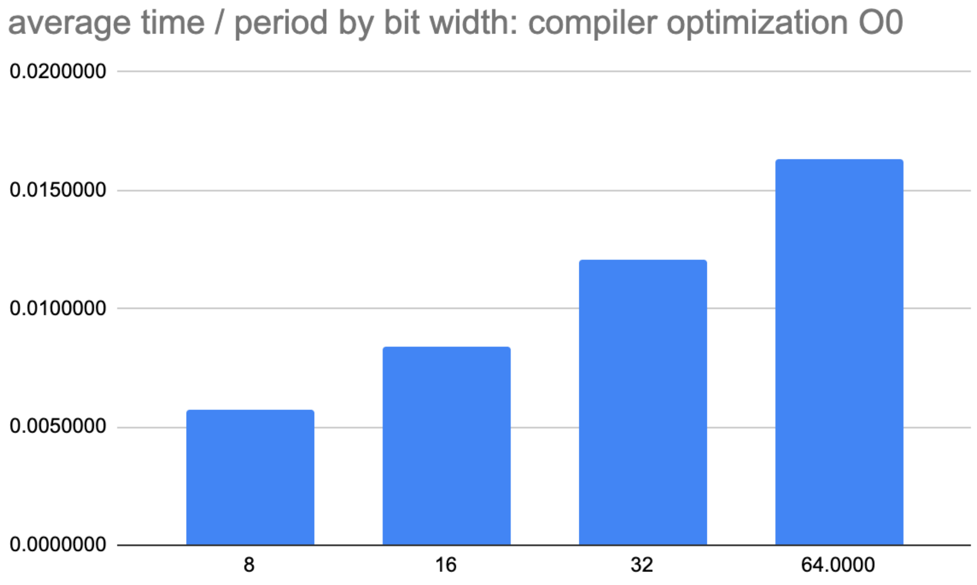


The time per period explodes as we start using more bits. After some heavy investigation, we found that the 64-bit parallelization didn't give the same speedup that followed the trend of the other versions of our method. In particular, each cycle was 5x slower on average.

Our first observation, from experimental data analysis, is that this is caused by a statistical phenomenon where the average of ratios is much greater than the ratio of averages. So this phenomenon is driven by tail events. The average of the time per period for 64-bit SIMD is 5x as long as the average time per period of 32-bit SIMD. However, the average time of 32-bit and 64-bit SIMD is 1.28 and 1.22 seconds respectively, and the average periods are 517 and 333 respectively, so the ratio of the averages is only about 1.5x, much less than the 5x driven by the tail events.

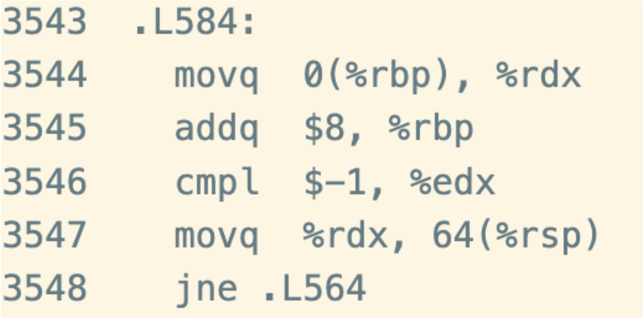
To try to conduct further analysis, we realized that 64-bit variables don't allow for register optimization nearly as much as any other variables. While the ALU operations on a x86/64 processor take the same time with 32 bits as with 64 bits, moving the variables in the registers becomes more complicated. 64 bit variables induce more register pressure during the unit_propagation and the compiler needs to spill more often. We even discovered some spilling inside the while loops. We investigated this by experimenting with compiler optimizations and by analyzing assembly code.

We first simply re-ran our experiments with gcc optimization level 0 instead of level 3, and found that the discrepancy is much smaller without register allocation.

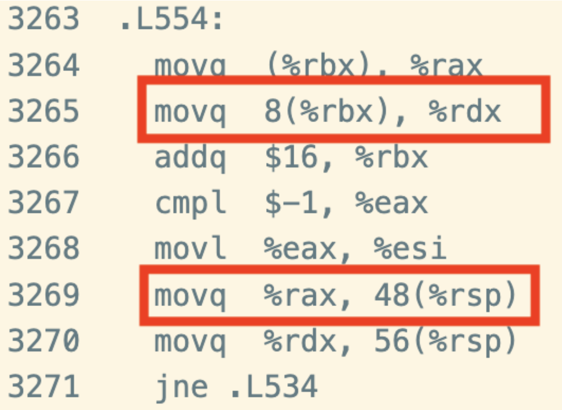


With empirical findings, we inspected assembly code directly. We specifically inspected the code for unit propagation, the most important function of our codebase, and found register spilling

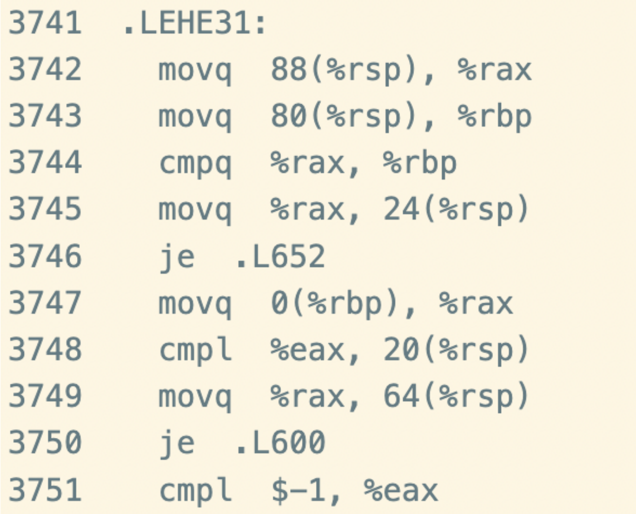
We first found, using inspection with tools like `wc, diff, cat, grep`, that the assembly was very similar for all SIMD widths besides 64-bit. We then inspected the changes further and found evidence of register spilling.



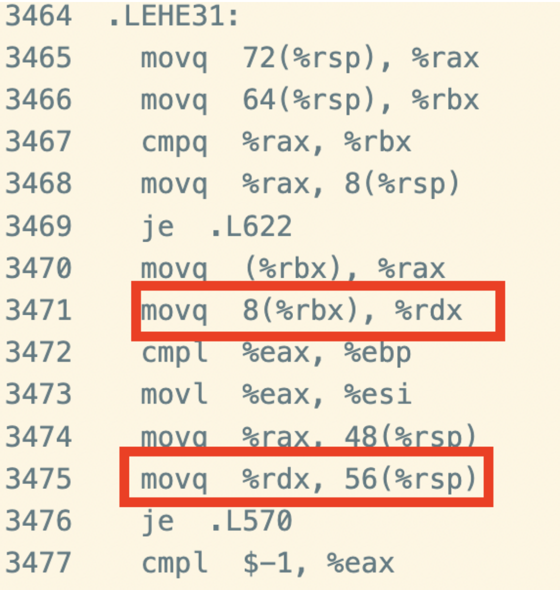
```
.L584:
  movq   0(%rbp), %rdx
  addq   $8, %rbp
  cmpl   $-1, %edx
  movq   %rdx, 64(%rsp)
  jne .L564
```



```
.L554:
  movq   (%rbx), %rax
  movq   8(%rbx), %rdx
  addq   $16, %rbx
  cmpl   $-1, %eax
  movl   %eax, %esi
  movq   %rax, 48(%rsp)
  movq   %rdx, 56(%rsp)
  jne .L534
```



```
.LEHE31:
  movq   88(%rsp), %rax
  movq   80(%rsp), %rbp
  cmpq   %rax, %rbp
  movq   %rax, 24(%rsp)
  je    .L652
  movq   0(%rbp), %rax
  cmpl   %eax, 20(%rsp)
  movq   %rax, 64(%rsp)
  je    .L600
  cmpl   $-1, %eax
```

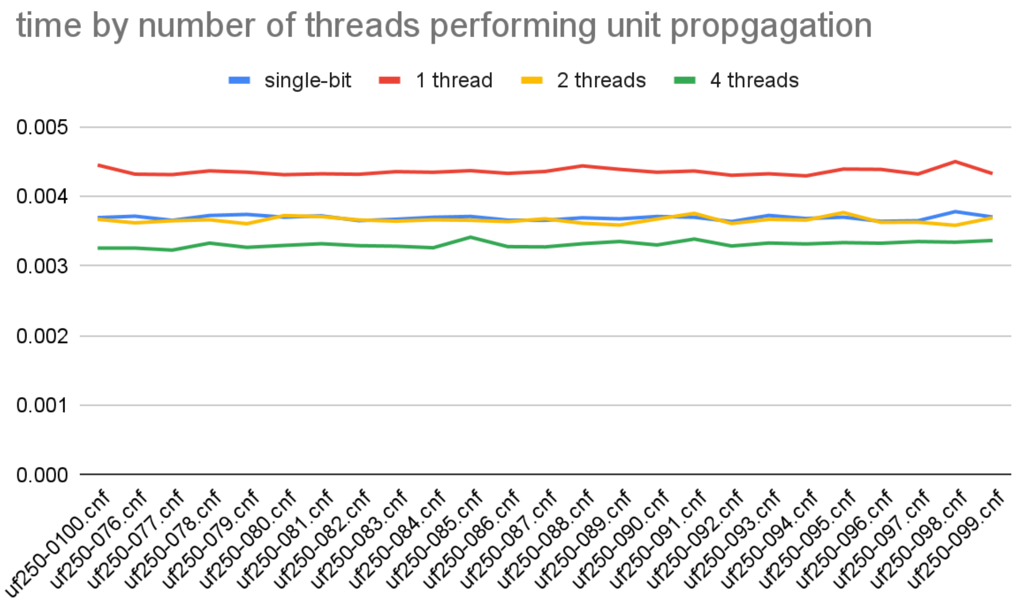


```
.LEHE31:
  movq   72(%rsp), %rax
  movq   64(%rsp), %rbx
  cmpq   %rax, %rbx
  movq   %rax, 8(%rsp)
  je    .L622
  movq   (%rbx), %rax
  movq   8(%rbx), %rdx
  cmpl   %eax, %ebp
  movl   %eax, %esi
  movq   %rax, 48(%rsp)
  movq   %rdx, 56(%rsp)
  je    .L570
  cmpl   $-1, %eax
```

The code snippets represented above represent two examples of a code portion where register spilling can be found with 64 bit vectors but not 32 bit vectors. The 32-bit assembly is on the left and the 64-bit assembly is on the right. This turns out to be the largest limitation on the data parallelism. It would be interesting to see what happens when data parallelism is increased even more (e.g. 512 bits). Does the reduction in the number of periods still make up for the increased time spent in each period?

Parallel Unit Propagation

We decided to test our OpenMP parallel unit propagation function with different thread numbers, run it on the GHC cluster (containing 8 x86/64 cores) and compare it against our original single-bit implementation of the SAT solver. Here are the results we got:



There are a few interesting things to note here:

- The 1-thread version of our parallel program is worse than the baseline (the single-bit version). This can be explained easily by the fact that we modified the original code in order to make it parallelizable (on different levels), thereby compromising the performance a little bit. Recall that the single-bit version of the code is not simply the multi-threaded code with 1 thread, but the most efficient possible single-threaded code. In this case, the unit_propagation was separated into two steps (the reading and the writing steps), whereas the single-threaded code of the baseline does both at the same time.

- The speedup is not as good as we expected. This can be explained by two reasons. First, Amdahl's law says that the speedup is bounded by the sequential portion of the code. Here, the sequential portion of the code is writing to phi_active. Recall that we chose to sequentialize the "write" part of unit_propagation in order to avoid false sharing. The other reason is that the workload is not very balanced. Each thread is responsible for a chunk of the remaining literals, and responsible for the chunk of the next iteration generated by the current chunk. We anticipated that the workload would be unbalanced and chose an OpenMP dynamic schedule, but apparently it does not entirely solve the problem entirely.

Parallel Duplicate Removal

The duplicate removal turns out to have almost no impact on the time needed by the algorithm. This was to be expected as it runs almost entirely in the background and there is very little communication with the main thread. The only communication happens via our home-made concurrent blocking queue every 5 periods. We also noticed that the amount of duplicates that appear depends on the bit vector width and the number of variables in the input. More specifically, we noticed the largest number of duplicates when the number of variables was small and the bit vector was large.

List of work by each student and distribution of credit

Sacha Bartholme: 60%

- Came up with this idea for SAT solving
- Implemented duplicate removal
- Implemented single bit version
- Implemented sequential unit propagation
- Analyzed the multi-bit assignment speedups
- Analyzed the multi-threaded unit propagation speedups
- Wrote part of the report

David Noursi: 40%

- Implemented and analyzed synchronized unit propagation
- Implemented test generator
- Implemented test parser and found test cases
- Analyzed the problem of the slow 64-bit width
- Wrote part of the report
- Unfamiliar with the method at the beginning of the project
- Was sick for about a week

References

[1] Heule, Marijn & Maaren, Hans. (2008). Parallel SAT Solving using Bit-level Operations. JSAT. 4. 99-116. 10.3233/SAT190040.

[2] http://www.satcompetition.org/

[3] https://en.wikipedia.org/wiki/Unit_propagation

[4] Escamocher, G., O'Sullivan, B., & Prestwich, S. D. (2019). Generating difficult sat instances by preventing triangles. arXiv preprint arXiv:1903.03592.

[5] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[6] http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/www.andrew.cmu.edu/user/rang/index.html

[7] https://sites.google.com/view/parallel-sat-solver

[8] http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/www.andrew.cmu.edu/user/rang/finalWriteup.pdf

[9] https://en.wikipedia.org/wiki/Circuit_satisfiability_problem