SAT Solver

Kevin, Jonathan, Max

What is a CNF expression?

All of the following formulas in the variables A, B, C, D, E, and F are in conjunctive normal form:

- $(A \vee \neg B \vee \neg C) \wedge (\neg D \vee E \vee F \vee D \vee F)$
- $(A \vee B) \wedge (C)$
- $(A \vee B)$
- (A)

The following formulas are **not** in conjunctive normal form:

- $\neg(A \wedge B)$, since an AND is nested within a NOT
- $\neg(A \vee B) \wedge C$, since an OR is nested within a NOT
- $A \wedge (B \vee (D \wedge E))$, since an AND is nested within an OR

Our Goal

Find a configuration for the variables that satisfy the expression.

Prove no configuration will ever solve the expression

```
p cnf 8 9
1 2 3 4 5 6 7 8 0
-1 -2 -3 -4 -5 0
1 -2 3 -4 5 0
5 7 -8 0
1 6 -4 0
-1 2 0
1 -2 0
1 2 0
-1 2 0
```

```
p cnf 8 9
1 2 3 4 5 6 7 8 0
-1 -2 -3 -4 -5 0
1 -2 3 -4 5 0
5 7 -8 0
1 6 -4 0
-1 2 0
1 -2 0
1 2 0
-1 -2 0
```

```
DPLL Solver result: UNSAT
```

DIMACS Format

```
DPLL Solver result: SAT: [Just True,Just True,Nothing,Just False,Nothing,Just True,Just True,Nothing
```

Naive Attempt

-We developed a simple naive algorithm that generates all possible configurations and checks whether each satisfies the given boolean expression. Being a brute force solution, this implementation was (expectedly) not very efficient.

p	q	z	((p ∨ q) ∧ ((q ∨ z) ∧ (z ∨ p)))
F	F	F	**F**
F	F	T	**F**
F	T	F	**F**
F	T	T	**T**
T	F	F	**F**
T	F	T	**T**
T	T	F	**T**
T	T	T	**T**

First try sequential

- We attempted to group numbers with similar binary encodings into equivalence classes, exploiting shared lower bits to simplify the Boolean expression using clause elimination, literal elimination, and early returns.
- This approach worked for small problems (e.g., 20 variables taking ~1 second), but became inefficient for larger ones (e.g., 50 variables taking >2 minutes) due to the need to check all numbers within each equivalence class.
- The method proved too slow, prompting us to pivot to a different approach.

For example:

- 0, 4, 8 all end in 00 or False, False
- 1, 5, 9 all end in 01 or False, True
- 2, 6, 10 all end in 10 or True, False
- 3, 7, 11 all end in 11 or True, True

Second Try Sequential (Best)

Rather than naively enumerating all possibilities of assignments then conducting a linear search over the possibilities, we can instead rely on our intuition when attempting to determine satisfiability by hand.

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

Second Try Sequential (Best)

$$\underbrace{(p_1 \vee \neg p_3 \vee \neg p_5)}_{C_1} \wedge \underbrace{(\neg p_1 \vee p_2)}_{C_2} \wedge \underbrace{(\neg p_1 \vee \neg p_3 \vee p_4)}_{C_3} \wedge \underbrace{(\neg p_1 \vee \neg p_2 \vee p_3)}_{C_5} \wedge \underbrace{(\neg p_4 \vee \neg p_2)}_{C_6}$$

Suppose we assign true to p_1. This leads to:

$$\begin{aligned} & (p_1 \vee \neg p_3 \vee \neg p_5) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\ \leftrightarrow\ & (\top \vee \neg p_3 \vee \neg p_5) \wedge (\bot \vee p_2) \wedge (\bot \vee \neg p_3 \vee p_4) \wedge (\bot \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\ \leftrightarrow\ & \top \wedge p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\ \leftrightarrow\ & p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \end{aligned}$$

Clause C_2, originally $\neg p_1 \vee p_2$, is now simply p_2. Thus, any satisfying interpretation must assign p_2=true. In this case, p_2 is called a “unit literal”. It occurs in a clause with no other literals.

Second Try Sequential (Best)

$$
\begin{aligned}
&(p_1 \vee \neg p_3 \vee \neg p_5) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\
\leftrightarrow\ &(\top \vee \neg p_3 \vee \neg p_5) \wedge (\bot \vee p_2) \wedge (\bot \vee \neg p_3 \vee p_4) \wedge (\bot \vee \neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\
\leftrightarrow\ &\top \wedge p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2) \\
\leftrightarrow\ &p_2 \wedge (\neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_4 \vee \neg p_2)
\end{aligned}
$$

Simplifying our formula further with p_2=true yields:

$$
\begin{aligned}
&\top \wedge (\neg p_3 \vee p_4) \wedge (\neg \top \vee p_3) \wedge (\neg p_4 \vee \neg \top) \\
\leftrightarrow\ &(\neg p_3 \vee p_4) \wedge (\bot \vee p_3) \wedge (\neg p_4 \vee \bot) \\
\leftrightarrow\ &(\neg p_3 \vee p_4) \wedge p_3 \wedge \neg p_4
\end{aligned}
$$

We again have two unit literals p_3 and $\neg p_4$. Thus, we must assign p_3=true.

Second Try Sequential (Best)

$$
\begin{aligned}
&\top \wedge (\neg p_3 \vee p_4) \wedge (\neg \top \vee p_3) \wedge (\neg p_4 \vee \neg \top) \\
\leftrightarrow\ &(\neg p_3 \vee p_4) \wedge (\bot \vee p_3) \wedge (\neg p_4 \vee \bot) \\
\leftrightarrow\ &(\neg p_3 \vee p_4) \wedge p_3 \wedge \neg p_4
\end{aligned}
$$

Simplifying our formula again with p_3=true yields:

$$
\begin{aligned}
&(\neg \top \vee p_4) \wedge \top \wedge \neg p_4 \\
\leftrightarrow\ &(\bot \vee p_4) \wedge \neg p_4 \\
\leftrightarrow\ &p_4 \wedge \neg p_4
\end{aligned}
$$

We are left with only clauses which are unit literals. This last formula, derived from the initial assignment p_1=true in the original formula, is unsatisfiable. So we cannot assign p_1=true in the original formula because of its implications. p_1=false, and we can follow similar “unit propagation” / BCP logic to determine the original formula is ultimately satisfiable (e.g. p_1=F, p_2=F, p_3=F, p_4=T, p_5=T/F)

Second Try Sequential (Best)

If this were implemented as a recursive algorithm, one recursive probe would reveal p_1=false. This is much better than having to linearly search through 2^5 assignments. We found the Davis-Putnam-Logemann-Loveland (DPLL) Algorithm, proposed in the 1960s, does exactly this BCP.

DPLL also performs Pure Literal Elimination: if a variable occurs with only one polarity in the formula, i.e. occurs only as a positive literal x or only as a negative literal ¬x, it is “pure”. Pure literals can be assigned a value such that all clauses containing it become true. Thus, clauses containing pure literals may also be removed from the formula along with those removed by BCP.

Second Try Sequential (Best)

```
Algorithm DPLL
    Input: A set of clauses Φ.
    Output: A truth value indicating whether Φ is satisfiable.
```

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.

Note:

- Termination conditions
- Flexibility in branching heuristic

Implications:

- Family of algorithms
- Chronological backtracking

First Parallel Attempt

For each variable we branched on both True and False configuration, using ‘par’ and ‘parseq’, sparking once using par for the false assignment and using parseq to evaluate the second assignment within the same parallel computation

```
(satFalse,falseAsgmt) `par` (satTrue,trueAsgmt) `pseq` if satTrue then (True, trueAsgmt) else if not satFalse then (False, V.empty) else (True,falseAsgmt)
```

```
(satFalse,falseAsgmt) = parDpll (d-1) tryFalseForm tryFalseAsg
(satTrue, trueAsgmt) = parDpll (d-1) tryTrueForm tryTrueAsg
```

First Parallel Attempt

-This approach did not work.

-The sparks were “dud,” not efficiently being used for parallel computation

-Trying different variations of our code did not work so we moved on to a different approach.

```
80,735,068,960 bytes allocated in the heap
 2,849,844,968 bytes copied during GC
     1,369,816 bytes maximum residency (560 sample(s))
       188,592 bytes maximum slop
            63 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     18915 colls, 18915 par    5.889s   2.687s     0.0001s    0.0007s
Gen  1       560 colls,   559 par    0.719s   0.276s     0.0005s    0.0008s

Parallel GC work balance: 5.51% (serial 0%, perfect 100%)

TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

SPARKS: 39471 (0 converted, 0 overflowed, 39471 dud, 0 GC'd, 0 fizzled)

INIT    time    0.004s  (  0.002s elapsed)
MUT     time   50.728s  ( 48.010s elapsed)
GC      time    6.608s  (  2.962s elapsed)
EXIT    time    0.002s  (  0.006s elapsed)
Total   time   57.341s  ( 50.980s elapsed)

Alloc rate    1,591,543,055 bytes per MUT second

Productivity  88.5% of total user, 94.2% of total elapsed
```

Second Parallel Attempt (Best)

Since we branch on True and False we can use a technique learned in class for dealing with pairs.

We can apply this to the earlier code shown.

```
satFalse = parDpll strat (d-1) tryFalseForm tryFalseAsg
satTrue  = parDpll strat (d-1) tryTrueForm tryTrueAsg
in specialOr ([satTrue, satFalse] `using` strat)
```

```
parPair :: Strategy (a,b)
parPair (a,b) = do
  a' <- rpar a
  b' <- rpar b
  return (a',b')
```

Second Parallel Attempt (Best)

As we can see this did much better. But there was still a lot of sparks being fizzled or GC'd. (this was for 150 variables)

Sequential result: 54.744 secs

Parallel result (best): 29.039

Parallel result (worst): 91.518

Speedup: 1.88

```
391,084,639,464 bytes allocated in the heap
 17,942,822,024 bytes copied during GC
      8,265,440 bytes maximum residency (609 sample(s))
        341,640 bytes maximum slop
             74 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0     13643 colls, 13643 par   21.547s   3.981s     0.0003s    0.0022s
 Gen  1       609 colls,   608 par    7.832s   0.871s     0.0014s    0.0018s

 Parallel GC work balance: 75.33% (serial 0%, perfect 100%)

 TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

 SPARKS: 1016 (29 converted, 0 overflowed, 0 dud, 62 GC'd, 925 fizzled)

 INIT    time    0.007s  (  0.004s elapsed)
 MUT     time  439.422s  ( 40.738s elapsed)
 GC      time   29.380s  (  4.852s elapsed)
 EXIT    time    0.206s  (  0.027s elapsed)
 Total   time  469.014s  ( 45.621s elapsed)

 Alloc rate    889,997,804 bytes per MUT second

 Productivity  93.7% of total user, 89.3% of total elapsed
```

Second Parallel Attempt (Best)

Too large (40 depth)

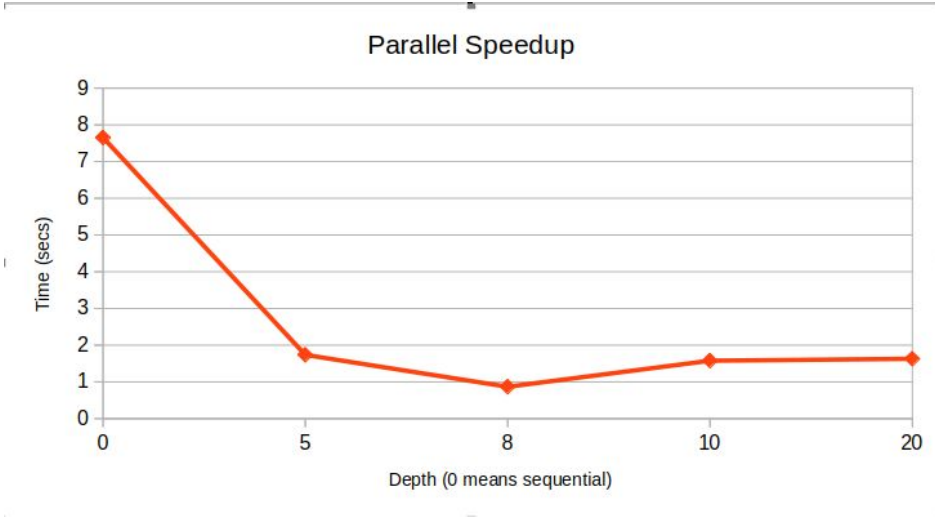
```
510,109,573,800 bytes allocated in the heap
 23,835,559,960 bytes copied during GC
      8,304,112 bytes maximum residency (824 sample(s))
        330,984 bytes maximum slop
             75 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0     18011 colls, 18011 par   28.755s   5.497s     0.0003s    0.0015s
 Gen  1       824 colls,   823 par   10.500s   1.201s     0.0015s    0.0022s

 Parallel GC work balance: 74.67% (serial 0%, perfect 100%)

 TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

 SPARKS: 497732 (41 converted, 0 overflowed, 0 dud, 142337 GC'd, 355354 fizzled)

 INIT    time    0.008s  (  0.004s elapsed)
 MUT     time  578.591s  ( 54.075s elapsed)
 GC      time   39.255s  (  6.699s elapsed)
 EXIT    time    0.012s  (  0.002s elapsed)
 Total   time  617.866s  ( 60.780s elapsed)

 Alloc rate    881,640,688 bytes per MUT second

 Productivity  93.6% of total user, 89.0% of total elapsed
```

Too small (10 depth)

```
394,742,930,024 bytes allocated in the heap
 18,348,805,032 bytes copied during GC
      8,977,304 bytes maximum residency (621 sample(s))
        331,344 bytes maximum slop
             77 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0     13892 colls, 13892 par   22.057s   4.202s     0.0003s    0.0012s
 Gen  1       621 colls,   620 par    8.002s   0.903s     0.0015s    0.0020s

 Parallel GC work balance: 74.82% (serial 0%, perfect 100%)

 TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

 SPARKS: 317076 (166 converted, 0 overflowed, 0 dud, 79857 GC'd, 237053 fizzled)

 INIT    time    0.007s  (  0.003s elapsed)
 MUT     time  444.274s  ( 41.266s elapsed)
 GC      time   30.058s  (  5.105s elapsed)
 EXIT    time    0.190s  (  0.026s elapsed)
 Total   time  474.529s  ( 46.400s elapsed)

 Alloc rate    888,512,887 bytes per MUT second

 Productivity  93.6% of total user, 88.9% of total elapsed
```

Second Parallel Attempt (Best)

Unsatisfiable Problems (100 variables, 430 clauses, 5 random samples):

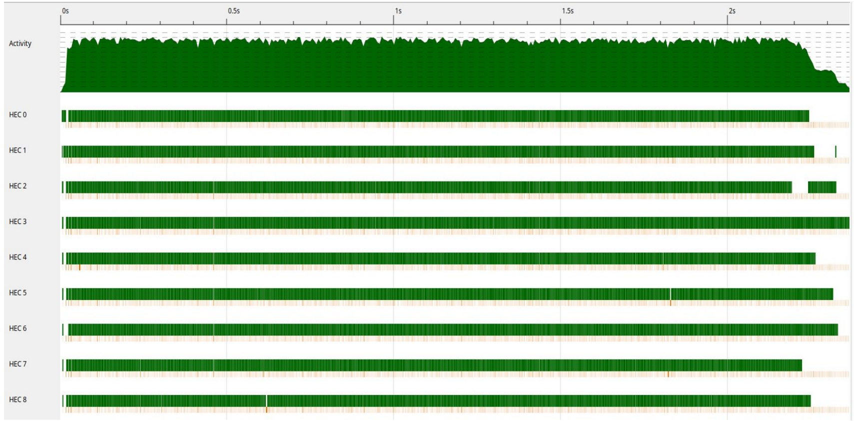


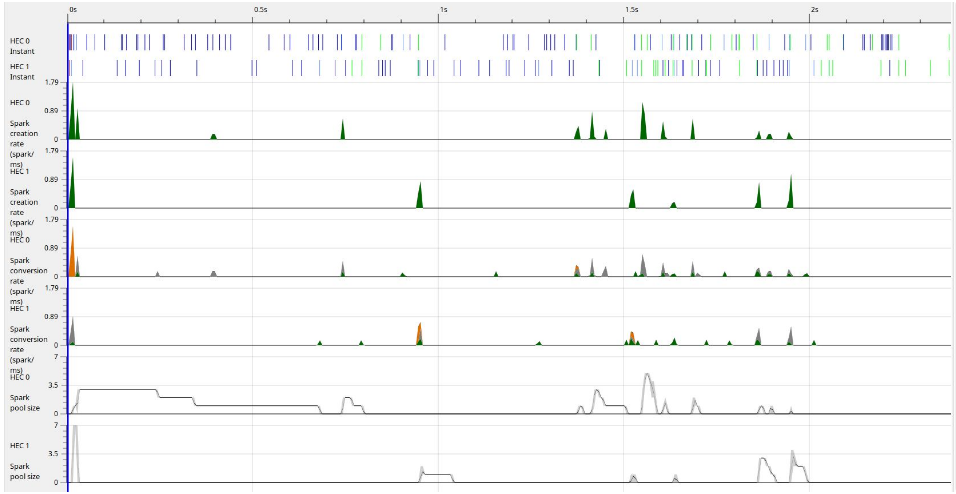

```
equential DPLL Solver result: UNSAT
 22,236,104,248 bytes allocated in the heap
    701,733,048 bytes copied during GC
      4,762,712 bytes maximum residency (54 sample(s))
        202,464 bytes maximum slop
             64 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0       609 colls,   609 par    0.929s   0.160s     0.0003s    0.0010s
 Gen  1        54 colls,    53 par    0.273s   0.038s     0.0007s    0.0013s

 Parallel GC work balance: 73.17% (serial 0%, perfect 100%)

 TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

 SPARKS: 488 (100 converted, 0 overflowed, 0 dud, 16 GC'd, 372 fizzled)

 INIT    time    0.004s  (  0.002s elapsed)
 MUT     time   24.110s  (  2.175s elapsed)
 GC      time    1.202s  (  0.197s elapsed)
 EXIT    time    0.001s  (  0.006s elapsed)
 Total   time   25.317s  (  2.381s elapsed)

 Alloc rate    922,288,082 bytes per MUT second

 Productivity  95.2% of total user, 91.4% of total elapsed
```

Second Parallel Attempt (Best)

Threadscope for best depth:
5.299 speedup

Sequential: Average: 7.652, STD DEV: 3.644
Best Parallel: Average: 1.444, STD DEV: .626

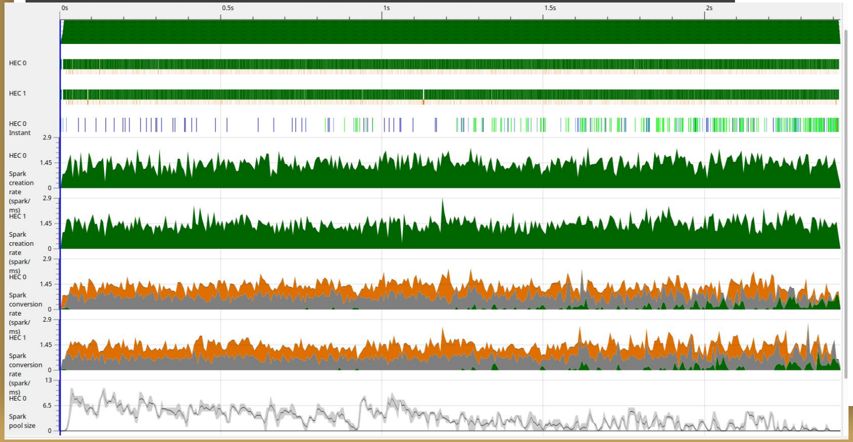




Second Parallel Attempt (Best)

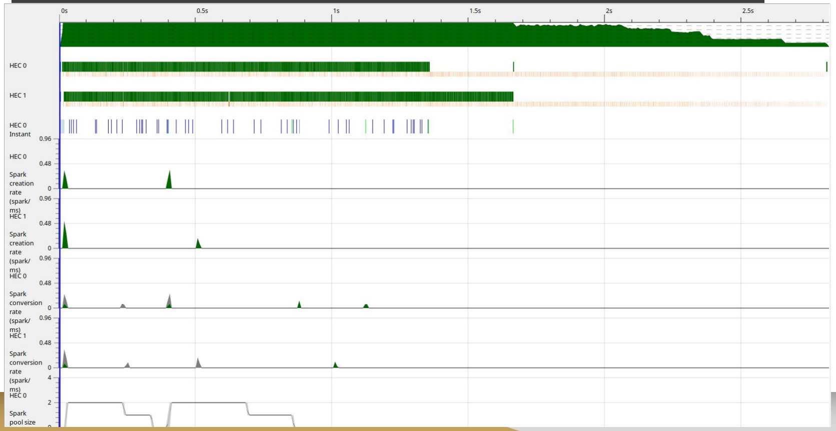
Too Large (20):

```
Sequential DPLL Solver result: UNSAT
  22,902,455,832 bytes allocated in the heap
     707,573,048 bytes copied during GC
       4,688,880 bytes maximum residency (54 sample(s))
         214,712 bytes maximum slop
              65 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause   Max pause
  Gen  0       609 colls,   609 par    0.921s   0.174s     0.0003s     0.0008s
  Gen  1        54 colls,    53 par    0.258s   0.039s     0.0007s     0.0010s

  Parallel GC work balance: 70.11% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 42226 (836 converted, 0 overflowed, 0 dud, 20338 GC'd, 21052 fizzled)

  INIT    time    0.010s  (  0.005s elapsed)
  MUT     time   24.886s  (  2.210s elapsed)
  GC      time    1.179s  (  0.213s elapsed)
  EXIT    time    0.002s  (  0.002s elapsed)
  Total   time   26.077s  (  2.430s elapsed)

  Alloc rate    920,280,096 bytes per MUT second
```



Too Small (5):

```
Sequential DPLL Solver result: UNSAT
  22,231,960,432 bytes allocated in the heap
     711,084,944 bytes copied during GC
       4,501,488 bytes maximum residency (69 sample(s))
         194,736 bytes maximum slop
              63 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause   Max pause
  Gen  0       936 colls,   936 par    0.864s   0.188s     0.0002s     0.0007s
  Gen  1        69 colls,    68 par    0.242s   0.041s     0.0006s     0.0011s

  Parallel GC work balance: 59.53% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 70 (21 converted, 0 overflowed, 0 dud, 8 GC'd, 41 fizzled)

  INIT    time    0.011s  (  0.005s elapsed)
  MUT     time   20.400s  (  2.562s elapsed)
  GC      time    1.107s  (  0.229s elapsed)
  EXIT    time    0.002s  (  0.004s elapsed)
  Total   time   21.519s  (  2.800s elapsed)

  Alloc rate    1,089,810,229 bytes per MUT second

  Productivity  94.8% of total user, 91.5% of total elapsed
```

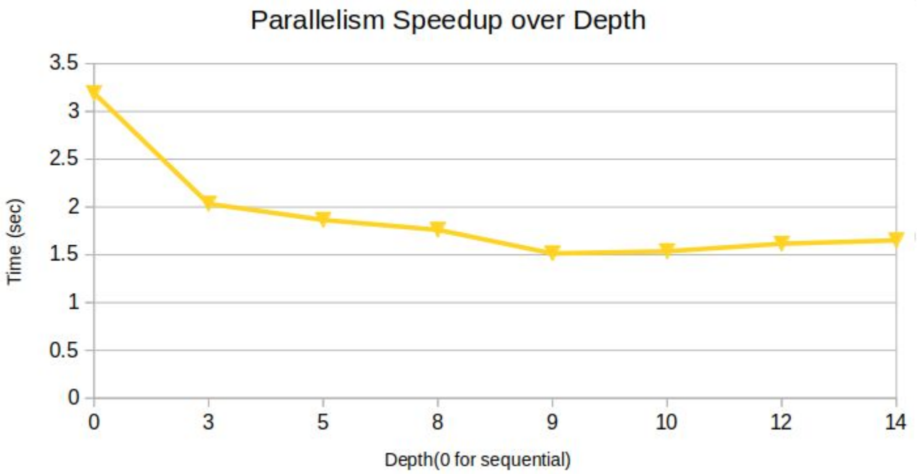


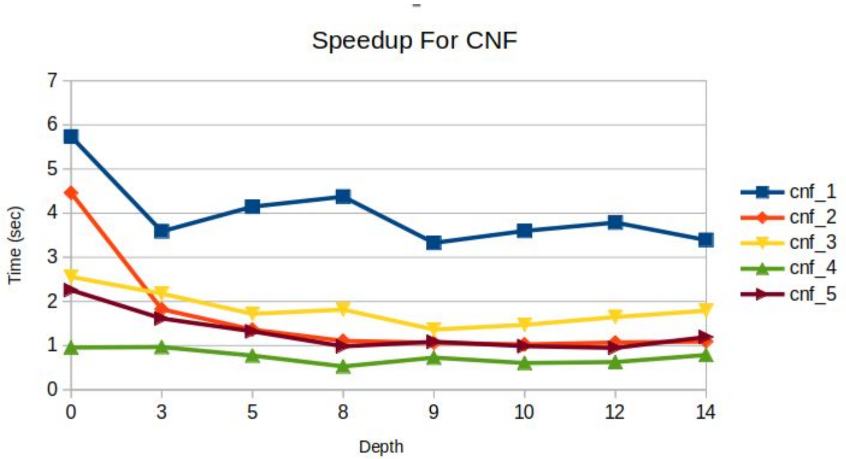
Second Parallel Attempt (Best)

Satisfiable 5 random cnf examples 100 variables 430 clauses

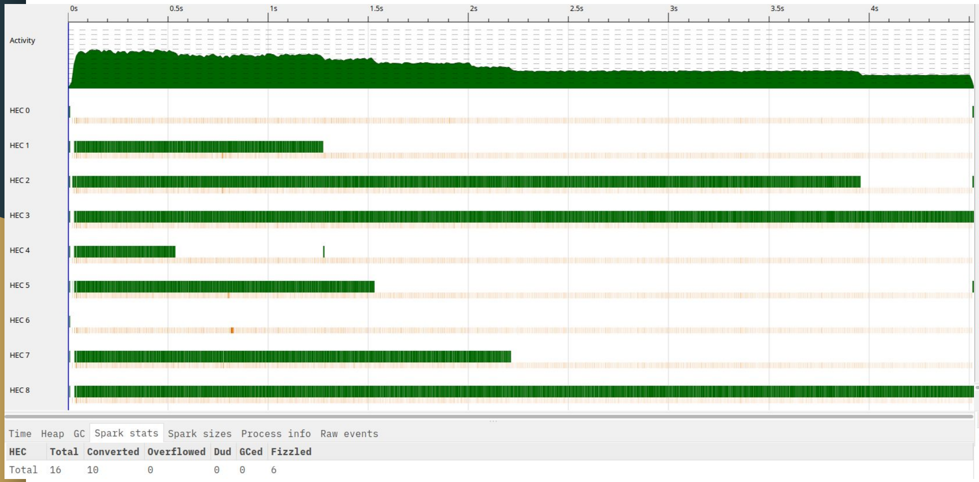
Average: 3.193 STD DEV: 1.864

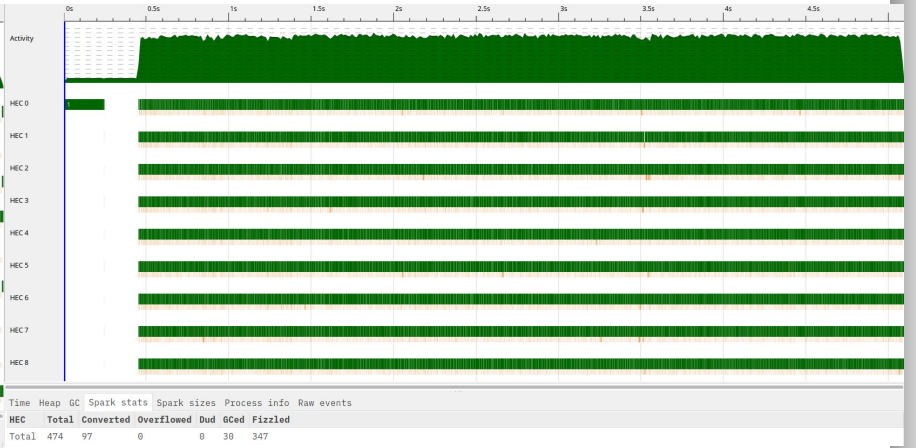
Best: 1.514 STD DEV: 1.038





Second Parallel Attempt (Best)





Third Parallel Attempt

We were able to add a top-level k-split based on variable frequency. Given a command-line argument, k, following the input file path, 2^k disjoint subproblems are created, where each subproblem represents a specific combination of true/false assignments for the k most frequent variables. These disjoint subproblems were then evaluated in parallel, each starting with a partial assignment for the k variables. Within each subproblem, the previous depth-limited parallel implementation explored the remaining search space by simultaneously assigning true/false for the current variable under consideration at a given depth, selected naively just as before.

Initial results were promising, but untuned, so we opted to stick with the previous depth-limited implementation for testing. E.g., UF150.645.100/uf150-01.cnf with depth 25 and 14 threads would be solved by the previous implementation in around 29 seconds, whereas the same file with depth 1024 and 14 threads was solved by this implementation in around 3.3 seconds (roughly 8.8x improvement).

Takeaways/Future

Haskell is hard.

If we were to continue on this problem we would look into finishing the parallel implementation for multiple assignments and sparking these using a "parlist" or possibly some other technique for condensing the number of sparks we generate.

We would also look into different heuristics and additional rules to improve the sequential algorithm, as well as different data structures to hold the boolean expressions in a more efficient manner to determine if a given configuration satisfies it or not.