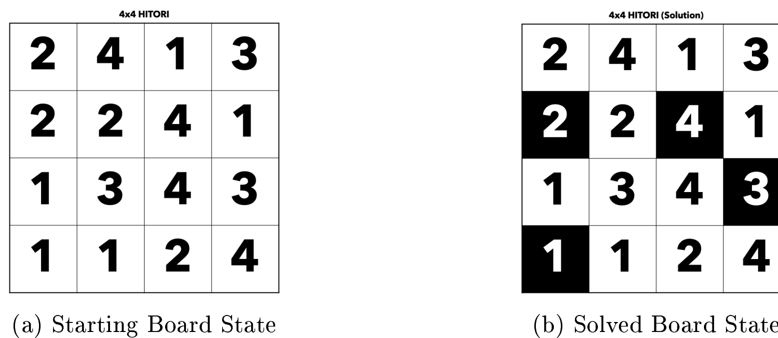
Hitori Final Report

Peter Yao (pby2101) Ava Hajratwala (ash2261)

Introduction to Hitori

Hitori is a single-player logic puzzle game that is NP-complete. The initial setup of the puzzle is a grid of numbers, where the player has to shade in specific cells to satisfy three rules: (1) No row or column may contain the same unshaded number more than once (2) shaded cells cannot lie adjacent to other shaded cells, although diagonals are allowed, and (3) all unshaded cells in the solved puzzles must be orthogonal to one another (there can be no unshaded "islands"). Here is an example of an unsolved puzzle and the solved solution[1]:



(a) Starting Board State (b) Solved Board State

Figure 1: Hitori Boards

Hitori Solver in Haskell

Most implementations of Hitori solvers use a breadth-first-search or depth-first-search approach. We take a slightly different angle and convert the Hitori instance to a solvable Boolean constraint problem (BCP). To do this, we must add logic to confirm that all three rules of Hitori are satisfied in conjunction.

To implement rule (1), we must ensure that no row or column contains an unshaded number more than once. For example, given a starting board, and a column or row vector, we could have $X = \{x_1, x_2, ..., x_n\}$ and we must have a rule such that for every pair $(x_i, x_j)_{i \neq j} \in X, x_i = x_j \Rightarrow \neg x_{i,shaded} \vee \neg x_{i,shaded}$. We implement this in Haskell by taking each column and row as a vector, identifying all pairs of elements, and for each pair, if the values are equal, we add a rule that at least one must be shaded to the CNF formula.

Similarly, for rule (2), we must ensure that any adjacent cells are not both shaded. So, for every x_i where x_j is an adjacent neighbor, we have $x_{i,unshaded} \vee x_{j,unshaded}$. We implement this in Haskell in a similar way to rule (1).

Finally, rule (3) is a bit trickier to implement. We borrow an idea from converting graph connectedness to a Boolean SAT problem.[2]. To begin, we know that at least one of the first two cells must be unshaded, due to rule (2). If the first cell is shaded, then the second cell (which is adjacent) must be unshaded. From here, there are two approaches to graph connectedness. We know that a graph is connected $\Leftrightarrow$ there exists a path of length i from every node to the starting unshaded node. However, given V vertices in a graph, this creates $|V| * i$ number of additional boolean

[1]Source: `https://baileyspuzzles.com/how-to-play-hitori/`

[2]Graph connectedness: `https://cs.stackexchange.com/questions/159983/boolean-constraints-for-a-connected-component-of-a-graph`

variables that the SAT solver would need to account for. For larger Hitori boards, this created complex SAT formulas that took a long time to implement.

Instead, we take the alternate approach, where we check that the graph is not disjoint, where a graph $G = (V, E)$ is disjoint $\Leftrightarrow$ given an arbitrary starting vertex $v_0 \in V$, for every $v \in V$, we have the following in conjunction:

- x_{v_0}
- $(\neg x_u \vee x_v) \wedge (x_u \vee \neg x_v) \forall (u, v) \in E$
- $\bigvee_{v \neq v_0} \neg x_v$

We can find the negation of this to return the formula that checks for disjointness, and only use $|V|$ additional variables in our final Boolean SAT. There was one caveat where if a vertex contains no valid edges, it is possible to set that "connectedness" variable to an arbitrary `True` or `False`, even though it is clearly disjoint. We add additional logic to ensure that each vertex contains at least one valid edge.

Boolean SAT Solvers

We found three types of Boolean SAT solvers in our research. We will briefly describe how each one works.

Davis-Putnam-Logemann-Loveland (DPLL)

DPLL is the most basic SAT solving algorithm that we worked with. It assumes that the input formula is in Conjunctive Normal Form, meaning formulas must be the ANDs of ORs. As a basic example:

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_1 \vee x_2)$$

A sequential DPLL SAT solver works in three phases:

1. Choose a literal to assign and assign it a value
2. Simplify the formula with that assignment
3. Recursively check if the resulting formula is satisfiable

If the recursive check for satisfiability succeeds, then the formula is satisfiable. Otherwise, the algorithm goes back to the previous guess and flips the assignment. This is called backtracking. Every time the algorithm makes a variable assignment, it can simplify the resulting formula in one of two ways: unit propagation and pure literal elimination. Unit propagation is the idea that if a clause only has one unassigned variable, the entire formula is only satisfiable if the literal tied to that variable is true. So we can take this assignment and propagate it down the tree. In pure literal elimination, the algorithm looks for variables with only one polarity in the entire formula. If a literal is pure, we know that it's possible to assign it such that all clauses containing the pure literal are true, so we remove all clauses containing pure literals.

Conflict Driven Clause Learning (CDCL)

The CDCL algorithm is based on the DPLL algorithm, which takes a literal and assigns a truth value to propagate a tree. However, it makes the observation that when conflicts are identified, new information is learned. And so, when new information is learned, we can apply a new clause to the original CNF formula to improve the efficiency of other branches. However, because locally learned clauses do not apply to the overall CNF formula with all branches, the algorithm must take careful steps to ensure that clause learning is limited only to certain subtrees.

Finally, when a conflict is detected, the CDCL algorithm backtracks back to an earlier guess, not necessarily to the prveious guess. Unlike DPLL, backtracking is done in a non-chronological way. This makes it more difficult to parallelize with a divide-and-conquer approach.

Lookahead

Lookahead algorithms are also based on DPLL. However, instead of picking a literal arbitrarily at every step, it will try each variable by assigning a `True` or `False` value to it. It will then look at the resulting formula and measure how much the assignment reduced the formula. So, at every step of the tree propagation, the algorithm will try to pick the best literal to perform the split.

To measure reduction, we can use multiple techniques. One way is to count the number of clauses or free variables remaining. However, some sources that we've read mention that the number of *reduced clauses* might be better. These are clauses where some of the variables have been eliminated. We have implemented the reduced clause measure in our Lookahead step.

Parallelizing Strategies

Divide and Conquer

A divide-and-conquer approach to parallelizing a SAT solver involves splitting the solver's search space. Our parallel DPLL implementation "divides and conquers" by sparking the evaluation of the positive and negative branches in parallel. One major problem with this approach is that each branch of computation is not guaranteed to be well-balanced, which leads to less-than-ideal speedup for parallel DPLL. Still, parallelizing DPLL is a worthwhile task, and we can still achieve some speedup.

Portfolio

Portfolio solving is another technique for parallelizing Boolean SAT solvers. To achieve this, sequential solvers with different algorithms run in parallel, working on solving the same problem. However, if we run the same solver against the same problem, we would obviously be duplicating work for ourselves. One way to avoid this is to run diffent types of solvers. For example, a DPLL algorithm and a Lookahead solver can run together. Also, because CDCL implementations have random seeds to determine start points, some portfolio strategies simply run multiple CDCL solvers in parallel, but with different starting seeds.

We don't take this approach due to the amount of work that gets duplicated by the strategy. Additionally, Haskell does not have an easy way to return results once one is found. All solvers must fully evaluate before we can return a result. Instead, we consider an alternative that combines divide-and-conquer and portfolio strategy. However, we do use a similar approach in our next strategy.

Cube and Conquer

Cube and conquer combines both divide-and-conquer and portfolio solvers. Initially, divide and conquer is used to split the original problem into multiple sub problems (called cubes). This can be done with DPLL, but we have implemented it with Lookahead. Because we are not evaluating the entire formula, we can use the expensive lookahead step to make sure that the cube subproblems are reduced as much as possible. Once these cubes are found, we can apply a CDCL solver to each cube in parallel, similar to a portfolio approach.

However, unlike in the traditional portfolio method, we are not repeating as much work, since each SAT solver is working on a slightly different subproblem. But, we still run into the same issue where each solver must evaluate completely before we can return a result. The solver needs to balance the workload of each cube to ensure each subproblem is roughly equal difficulty.

Results

Here are some of the results from our testing. In the following pictures, 0 threads indicates the sequential algorithm, which we use as a baseline for speedup. The file we used was randomly selected from a database of CNF DIMACs files of randomly generated 3-SAT instances with 100 variables, 418 clauses, and a backbone size of 90 [3]. From this collection, we selected *CBS_k3_n100_m411_b90_999.cnf*. All tests were all conducted on a 2023 Apple MacBook Pro running Apple's M2 Pro chip with 12 cores. For each configuration, we took an average of 3 runs to determine the runtime.

- **Parallel Look-Ahead**

  For the parallel look-ahead algorithm implementation, we had the worst baseline runtime. However, we were able to get some improvements from running parallel lookahead along with parallel evaluation of the implication graph.

  Here are the results from our best settings. We can see better efficiency in spark conversion, which wasn't the case with greater parallelization depth thresholds. A full table of our results is below, along with our threadscope analysis.

```
(base) peteryao@Peters-MacBook-Pro Final Project \% time ./CNFtest +RTS -N11 -s
  12,911,601,136 bytes allocated in the heap
     107,002,904 bytes copied during GC
       1,306,864 bytes maximum residency (44 sample(s))
         157,416 bytes maximum slop
              60 MiB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed) Avg pause Max pause
  Gen  0       388 colls,  388 par    0.147s  0.052s    0.0001s   0.0012s
  Gen  1        44 colls,   43 par    0.041s  0.015s    0.0003s   0.0015s

  Parallel GC work balance: 50.57\% (serial 0\%, perfect 100\%)

  TASKS: 24 (1 bound, 23 peak workers (23 total), using -N11)

  SPARKS: 32895 (30038 converted, 0 overflowed, 0 dud, 1434 GC'd, 1423 fizzled)

  INIT    time    0.000s  (  0.011s elapsed)
  MUT     time    5.115s  (  0.629s elapsed)
  GC      time    0.188s  (  0.067s elapsed)
  EXIT    time    0.006s  (  0.012s elapsed)
  Total   time    5.309s  (  0.719s elapsed)

  Alloc rate    2,524,289,833 bytes per MUT second

  Productivity 96.3\% of total user, 87.5\% of total elapsed

./CNFtest +RTS -N11 -s 5.31s user 0.26s system 754\% cpu 0.739 total
```

[3] Singer, Josh. *CBS_k3_n100_m411_b90* https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Total Time (seconds)									
		Parallelization Depth Threshold							
		1	2	5	10	15	25	50	100
Threads	0	2.275	2.275	2.275	2.275	2.275	2.275	2.275	2.275
	1	2.289	2.293	2.342	2.356	2.341	2.273	2.279	2.341
	2	2.408	2.418	2.409	2.406	2.404	2.406	2.410	2.408
	3	2.226	2.765	2.825	2.485	2.816	2.820	2.484	2.490
	4	1.513	2.491	2.479	2.601	2.573	2.456	2.492	2.738
	5	1.516	1.943	2.113	2.290	1.810	2.040	1.970	2.297
	6	1.254	1.484	1.795	1.958	1.802	2.055	1.774	1.671
	7	0.949	1.484	1.626	1.631	1.611	1.383	1.476	1.575
	8	1.024	1.295	1.407	1.359	1.393	1.447	1.663	1.411
	9	0.966	1.462	1.534	1.308	1.664	1.444	1.748	1.484
	10	0.777	0.837	1.347	1.121	1.455	1.402	1.130	1.373
	11	0.744	1.010	1.180	1.382	1.140	1.234	1.459	1.455
	12	0.751	0.968	1.220	1.231	1.367	1.434	1.096	1.383

(a) Runtime in seconds

Speedup									
		Parallelization Depth Threshold							
		1	2	5	10	15	25	50	100
Threads	0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	1	0.994	0.992	0.971	0.966	0.972	1.001	0.998	0.972
	2	0.945	0.941	0.944	0.946	0.946	0.946	0.944	0.945
	3	1.022	0.823	0.805	0.916	0.808	0.807	0.916	0.914
	4	1.504	0.913	0.918	0.875	0.884	0.926	0.913	0.831
	5	1.500	1.171	1.077	0.993	1.257	1.115	1.155	0.991
	6	1.815	1.533	1.267	1.162	1.262	1.107	1.283	1.362
	7	2.396	1.533	1.399	1.395	1.412	1.645	1.541	1.445
	8	2.221	1.757	1.617	1.674	1.633	1.572	1.368	1.612
	9	2.354	1.556	1.483	1.740	1.367	1.575	1.301	1.533
	10	2.929	2.718	1.689	2.029	1.564	1.622	2.013	1.657
	11	3.059	2.252	1.927	1.647	1.996	1.844	1.559	1.563
	12	3.029	2.351	1.865	1.849	1.664	1.586	2.076	1.645

(b) Speedup factor

Figure 2: Parallel Look-Ahead Performance

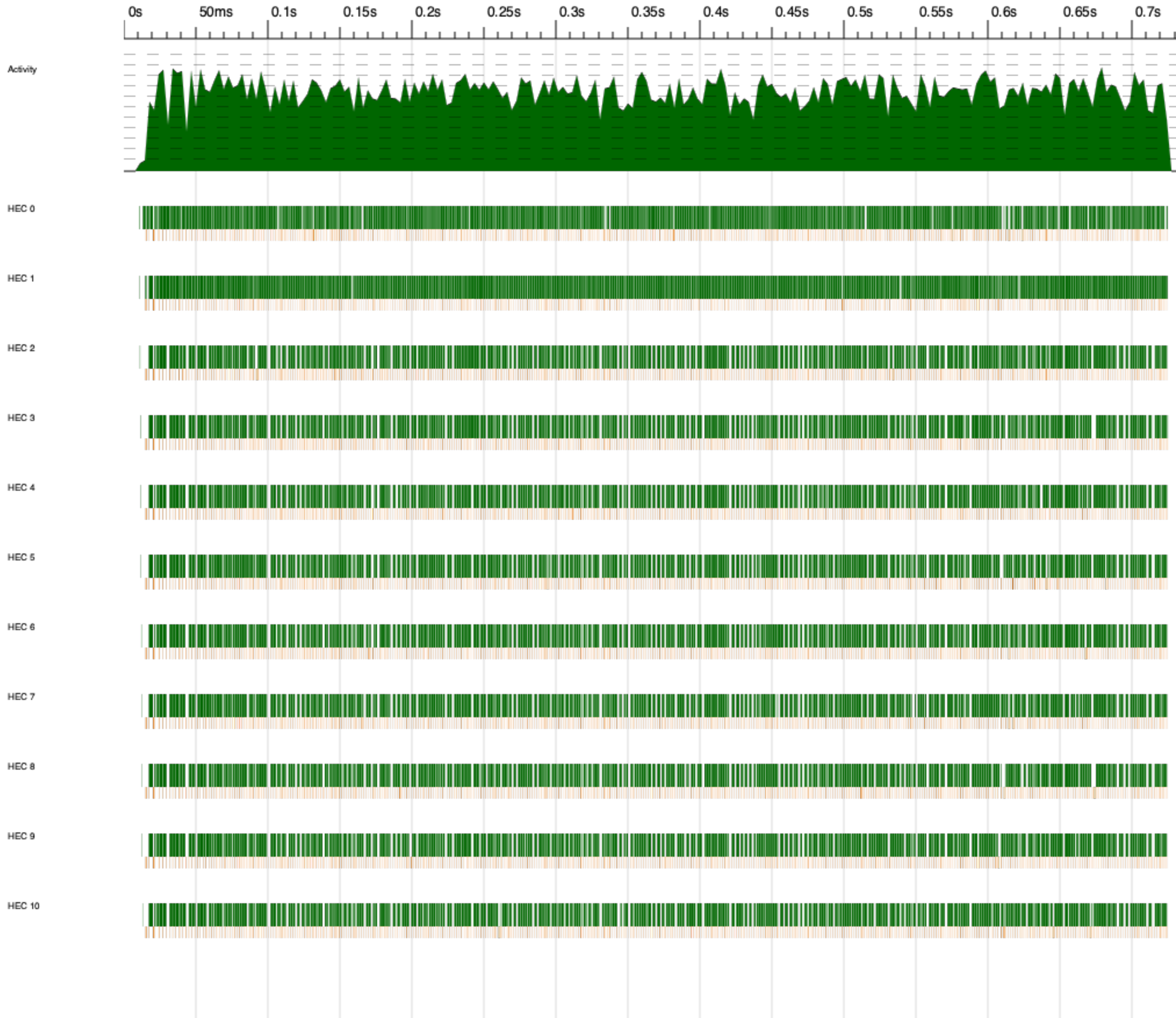


Figure 3: Parallel Look-Ahead Threadscope Analysis

- **CDCL Cube-and-Conquer** Our CDCL cube-and-conquer implementation suffers from a poor lookahead algorithm, and a relatively slow CDCL implementation. Although CDCL can outperform DPLL in many instances, the only existing implementation that we could find had an average runtime of 2.119s on our example formula. However, we noticed that just by breaking down the problem into cubes, CDCL was able to run very quickly, leading to strong runtimes even when using just 1 core.

  The best settings were generally used all cores, and the speedup was more apparent at greater look-ahead depths, as the problem approached perfect parallelism. Here are our results, including threadscope analysis of our best parameters.

```
(base) peteryao@Peters-MacBook-Pro Final Project \% time ./CNFtest +RTS -N11 -s
True
    699,163,104 bytes allocated in the heap
      6,336,216 bytes copied during GC
        981,984 bytes maximum residency (4 sample(s))
        135,672 bytes maximum slop
             59 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed) Avg pause Max pause
  Gen  0        24 colls,   24 par    0.009s   0.004s     0.0002s    0.0006s
  Gen  1         4 colls,    3 par    0.006s   0.002s     0.0006s    0.0013s

  Parallel GC work balance: 55.29\% (serial 0\%, perfect 100\%)

  TASKS: 24 (1 bound, 23 peak workers (23 total), using -N11)

  SPARKS: 696 (671 converted, 0 overflowed, 0 dud, 18 GC'd, 7 fizzled)

  INIT    time    0.000s  (  0.009s elapsed)
  MUT     time    0.512s  (  0.073s elapsed)
  GC      time    0.014s  (  0.006s elapsed)
  EXIT    time    0.011s  (  0.000s elapsed)
  Total   time    0.537s  (  0.088s elapsed)

  Alloc rate    1,366,342,854 bytes per MUT second

  Productivity  95.3\% of total user, 82.2\% of total elapsed

./CNFtest +RTS -N11 -s 0.54s user 0.04s system 116\% cpu 0.501 total
```

Total Time (seconds)							
		Lookahead Depth					
		0	1	2	3	4	5
Threads	0	2.119	2.119	2.119	2.119	2.119	2.119
	1	4.720	0.182	0.185	0.834	0.943	1.494
	2	4.709	0.172	0.135	0.611	0.538	0.972
	3	4.746	0.160	0.118	0.594	0.491	0.961
	4	4.729	0.147	0.108	0.582	0.477	0.831
	5	4.708	0.139	0.099	0.585	0.436	0.687
	6	4.785	0.135	0.086	0.591	0.375	0.585
	7	4.748	0.137	0.090	0.560	0.328	0.505
	8	4.748	0.135	0.092	0.562	0.289	0.440
	9	4.760	0.136	0.076	0.565	0.268	0.411
	10	4.775	0.137	0.075	0.567	0.256	0.390
	11	4.795	0.134	0.073	0.571	0.239	0.368
	12	4.817	0.136	0.076	0.576	0.228	0.356

(a) Runtime in seconds

Speedup							
		Lookahead Depth					
		0	1	2	3	4	5
Threads	0	1.000	1.000	1.000	1.000	1.000	1.000
	1	0.449	11.622	11.475	2.541	2.248	1.419
	2	0.450	12.344	15.696	3.468	3.939	2.179
	3	0.446	13.216	18.008	3.569	4.319	2.206
	4	0.448	14.415	19.560	3.639	4.439	2.549
	5	0.450	15.281	21.476	3.622	4.864	3.086
	6	0.443	15.658	24.544	3.583	5.656	3.624
	7	0.446	15.505	23.458	3.786	6.467	4.196
	8	0.446	15.658	23.033	3.768	7.332	4.816
	9	0.445	15.581	27.882	3.753	7.917	5.160
	10	0.444	15.467	28.128	3.737	8.277	5.433
	11	0.442	15.774	29.027	3.711	8.878	5.753
	12	0.440	15.581	27.882	3.679	9.307	5.952

(b) Cube factor

Figure 4: Parallel Cube-and-Conquer Performance

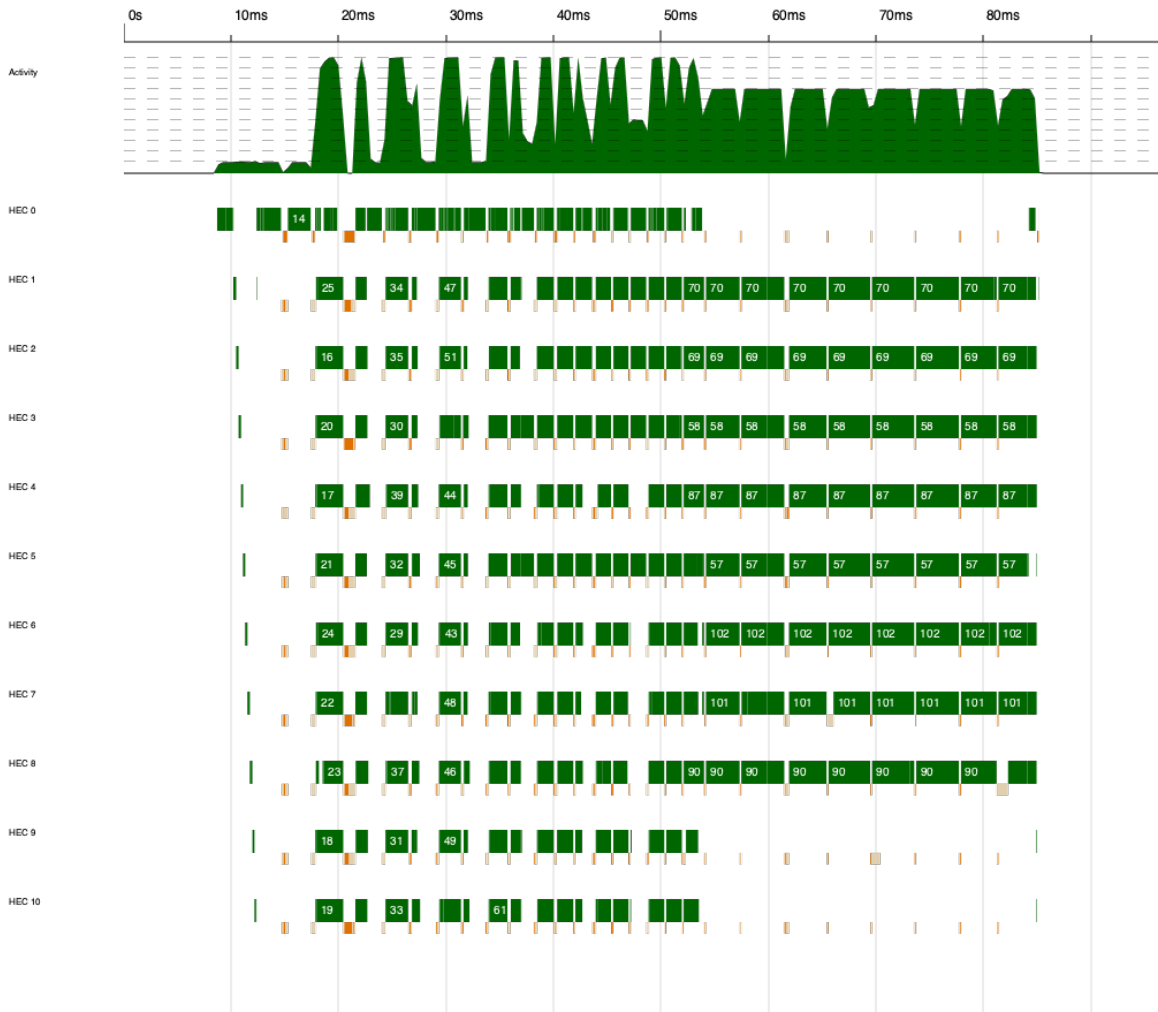


Figure 5: Parallel Cube-and-Conquer Threadscope Analysis

- **Parallel DPLL** The implementation we found for sequential DPLL was the fastest algorithm at solving our selected problem. However, unlike the parallel lookahead, the only parallel step in this implementation is when we explore the next layer of the implication graph. One key feature of our parallel DPLL approach is having control over a search "threshold" — that is, a global search depth after which we switch from a parallel algorithm to a sequential one. This is to minimize the overhead from parallelization when the clause set is very small. Since our spark threshold is connected to the depth of the implication graph, at a certain point, there is no additional change from increasing the threshold–everything always runs in parallel.

  Although the data is not listed in this report, we did notice that the performance increased significantly from `threshold = 6` to `threshold = 7`. In our tables below, we have `threshold = 5` and `threshold = 8` for spacing. The threadscope results look very similar to the Lookahead parallelization, with lots of time spent on garbage collection. Our results are listed below:

```
(base) peteryao@Peters-MacBook-Pro Final Project \% time ./CNFtest +RTS -N10 -s
   5,079,773,968 bytes allocated in the heap
     147,360,512 bytes copied during GC
       3,724,848 bytes maximum residency (14 sample(s))
         156,776 bytes maximum slop
              61 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed) Avg pause Max pause
  Gen  0       127 colls,  127 par    0.104s  0.031s    0.0002s   0.0010s
  Gen  1        14 colls,   13 par    0.039s  0.007s    0.0005s   0.0010s

  Parallel GC work balance: 69.33\% (serial 0\%, perfect 100\%)

  TASKS: 23 (1 bound, 21 peak workers (22 total), using -N10)

  SPARKS: 4610 (89 converted, 0 overflowed, 0 dud, 1986 GC'd, 2535 fizzled)

  INIT    time    0.000s  (  0.008s elapsed)
  MUT     time    1.252s  (  0.140s elapsed)
  GC      time    0.143s  (  0.039s elapsed)
  EXIT    time    0.131s  (  0.019s elapsed)
  Total   time    1.527s  (  0.206s elapsed)

  Alloc rate    4,056,394,350 bytes per MUT second

  Productivity 82.0\% of total user, 67.9\% of total elapsed

./CNFtest +RTS -N10 -s 1.53s user 0.08s system 733\% cpu 0.219 total
```

Total Time (seconds)									
		Parallelization Depth Threshold							
		1	2	5	8	10	20	25	50
Threads	0	0.290	0.290	0.290	0.290	0.290	0.290	0.290	0.290
	1	0.289	0.294	0.292	0.295	0.288	0.293	0.293	0.290
	2	0.294	0.303	0.304	0.305	0.299	0.304	0.305	0.299
	3	0.297	0.308	0.312	0.312	0.307	0.312	0.312	0.306
	4	0.298	0.313	0.313	0.312	0.309	0.313	0.313	0.308
	5	0.303	0.312	0.315	0.317	0.315	0.318	0.317	0.311
	6	0.306	0.313	0.320	0.227	0.252	0.250	0.286	0.282
	7	0.308	0.314	0.319	0.200	0.226	0.233	0.244	0.231
	8	0.310	0.313	0.320	0.167	0.168	0.163	0.161	0.224
	9	0.313	0.322	0.338	0.281	0.281	0.205	0.208	0.205
	10	0.318	0.326	0.369	0.201	0.221	0.170	0.192	0.143
	11	0.320	0.328	0.382	0.302	0.252	0.158	0.168	0.161
	12	0.322	0.330	0.408	0.213	0.162	0.173	0.195	0.226

(a) Runtime in seconds

Speedup									
		Parallelization Depth Threshold							
		1	2	5	8	10	20	25	50
Threads	0	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	1	1.003	0.986	0.993	0.982	1.006	0.988	0.988	0.999
	2	0.985	0.957	0.952	0.951	0.970	0.953	0.951	0.968
	3	0.974	0.940	0.929	0.928	0.944	0.929	0.928	0.946
	4	0.972	0.926	0.925	0.927	0.937	0.925	0.924	0.941
	5	0.955	0.927	0.920	0.915	0.921	0.912	0.913	0.930
	6	0.947	0.924	0.905	1.278	1.151	1.159	1.014	1.028
	7	0.941	0.923	0.907	1.448	1.284	1.241	1.187	1.252
	8	0.934	0.924	0.904	1.738	1.724	1.773	1.795	1.295
	9	0.924	0.901	0.856	1.030	1.030	1.415	1.390	1.415
	10	0.911	0.889	0.786	1.439	1.313	1.704	1.511	2.030
	11	0.905	0.883	0.758	0.958	1.148	1.837	1.724	1.795
	12	0.900	0.879	0.711	1.362	1.792	1.674	1.483	1.280

(b) Speedup factor

Figure 6: Parallel DPLL Performance

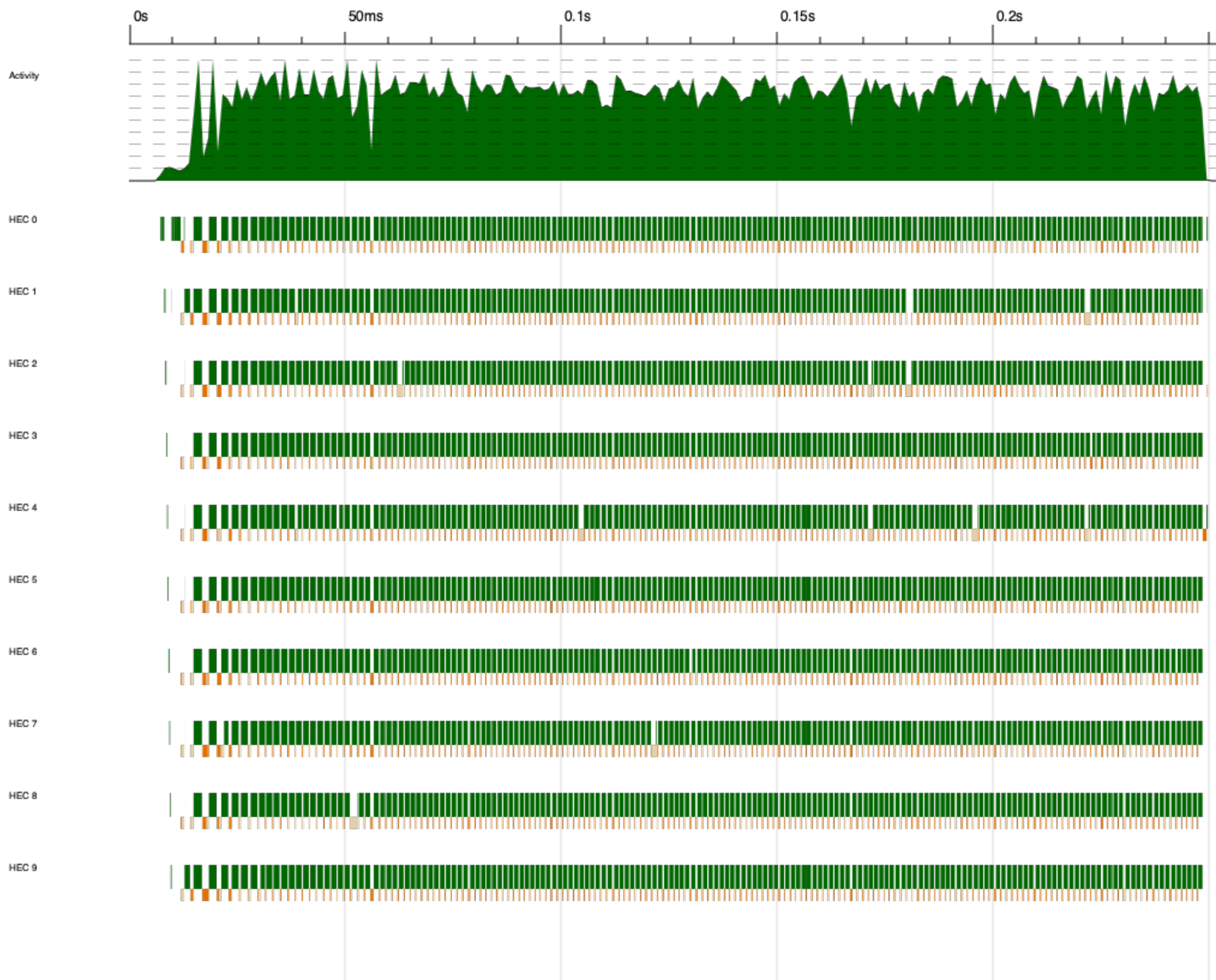


Figure 7: Parallel DPLL Threadscope Analysis

Conclusions

- **Lookahead vs Naive DPLL vs CDCL** Before testing, we expected CDCL to generally dominate DPLL on most problems. [4] However, this was not the case when we looked at existing implementations of CDCL and DPLL in Haskell. One of the best DPLL algorithms we found was able to solver our test instance in around 0.30s, while CDCL took around 2.1s. Lookahead performed the worst. When running lookahead logic on larger, unsatisfiable expressions, the expensive literal selection step added significantly to our runtime, even when parallelized.

  Part of the reason for the poor performance could be due to poor data structures and optimization. We were able to find several DPLL implementations, but only one CDCL implementation (not including MiniSAT, Mios, etc.). And we found no implementations for lookahead, so we had to write that ourselves.

- **Parallel DPLL** Our best speedup from using parallel DPLL came from using a high depth parameter and lots of cores. Under these conditions, there were a lot of sparks being created and a lot of cores potentially available to handle them. As expected, though, the speedup was far from the ideal speedup of 2x per core because of the load balancing problem inherent to the divide-and-conquer philosophy DPLL uses. With our best speedup (10 cores with threshold=50), 4610 sparks were created, of which 1986 were garbage collected. In other words, 1986 sparks weren't actually needed by the program. They likely represent the number of sparks created down one branch before the algorithm reached a satisfiable assignment on another branch.

- **Parallel Lookahead** We implemented pararllel lookahead in two steps. First, we parallelized the lookahead step, where we analyzed each variable's effect on problem reduction. Then, we used a divide and conquer strategy for the rest of the parallelization. We noticed the slowest performance between 3 to 5 threads. The best performance was around used nearly all of the 12 threads available. I would be curious to see if adding additional threads beyond 12 would improve or hurt the overall performance. Also, we decided to use spark creation scaling that was closer to logarithmic scale. Interestingly, the limits did not seem to have much effect on the overall performance, with the scaling and runtimes fairly consistent across testing.

- **Cube-and-Conquer** Here, we noticed the largest jump in performance when we ran our tests on our dataset. We decided to run this on our CDCL implementation so that the improvements would be more obvious. Sequential CDCL had an average performance of 2.119s in our test file. However, just by breaking the problem down, we were able to see massive increases in performance. Running cube-and-conquer on just one thread made tremendous gains. However, for our test file we did notice that as the problem broke down into smaller cubes, the runtimes got worse.

  As mentioned earlier, the more branching we do in the lookahead step, the more parallel work we can get from the original problem. We noticed that as the depths started to increase, we could see more of the thread scaling in effect. As the number of cubes increased, we had better performance as we added more cores to the problem. For smaller number of cubes, the effect is more muted, with similar performances across all runs.

  Theoretically, any SAT solver would be appropriate to use in a cube-and-conquer strategy, since the conquer step just assigns a solver to each cube. It's possible that with more efficient lookahead logic and a faster CDCL implementation, we could see even better results.

- **Back to Hitori** Looking back at our original Hitori problem, for 20x20 boards (among the largest that we could find), the CNF equations that we generate are not too difficult to solve. Any of these algorithms work just fine. Although one thing we can mention is that CNF rules generated from (1) No row or column containing the same unshaded number and (2) no adjacent shaded cells are always appended to the start of our CNF formula. These are also the easiest to solve, and help to limit the search space. Here, the expense of looking at every literal in a lookahead step might was not worth any potential benefits. CDCL was able to solve a 20x20 instance in 0.042s, while Cube-and-Conquer required around 0.50s under the best parameters.

[4] *https://cs.stackexchange.com/questions/3014/running-time-of-cdcl-compared-to-dpll*

Future Work

In future work on this project, we would want to consider clause sharing and its effect on our runtimes. When running a strategy like cube-and-conquer, some work is repeated by each solver, as it tries to evaluate subproblems with mostly the same variables. However, it is possible for the solvers to each work individually on their own problem, but then reconvene occasionally and share results with other solvers.

With more time, we could also improve our look-ahead implementation to get better cubes and find them more efficiently. One alternative for larger problems is to only consider a selection of literals to evaluate for clause reduction. Also, our look-ahead implementation was based DPLL, so we terminate look-ahead after a certain depth has been reached. This can create a lot of variability in the complexity of the cube required for the SAT solver to evaluate. One way to get around this is to purely run sequential CDCL for smaller problems, where cubing is not needed. And for larger problems, reduce the original formula until a certain level of reduction is achieved in each cube.

Additionally, the CDCL implementation that we found was relatively slow for our selected problem. However, it did perform faster on our Hitori formula. DPLL took 0.104s compared to 0.042 on an average of 3 runs. We could either adapt our cube-and-conquer to an improved CDCL algorithm, or pair with a known fast solver like MiniSAT.

Code

All code is located in our GitHub repository: `https://github.com/peterbyao/HitoriSolver/`

References

- Hitori rules and background `https://baileyspuzzles.com/how-to-play-hitori/`
- Puzzle database: `https://menneske.no/hitori/eng/`
- van der Knijf, Gerhard.*Solving and Generating Puzzles with a Connectivity Constraint* `https://www.cs.ru.nl/bachelors-theses/2021/Gerhard_van_der_Knijff___1006946___Solving_and_generating_puzzles_with_a_connectivity_constraint.pdf`
- D.W. *Boolean Constraints for a Connected Component of a Graph* `https://cs.stackexchange.com/questions/159983/boolean-constraints-for-a-connected-component-of-a-graph`
- Yuval Filmus. *SAT Algorithm for Determining if a Graph is Disjoint* `https://cs.stackexchange.com/questions/111410/sat-algorithm-for-determining-if-a-graph-is-disjoint?rq=1`
- Gander, Matthias. *Hitori Solver* `https://pdfcoffee.com/download/hitori-pdf-free.html`
- Juho. *Running time of CDCL compared to DPLL* `https://cs.stackexchange.com/questions/3014/running-time-of-cdcl-compared-to-dpll`
- Berger, Till and David Sabel *Parallelizing DPLL in Haskell* `https://www2.ki.informatik.uni-frankfurt.de/papers/sabel/berger-sabel-13.pdf`
- Gatlin. *DPLL* `https://gist.github.com/gatlin/1755736`
- Tamionv. *CDCL* `https://github.com/tamionv/HaskSat/tree/master`
- Huele, Marijn. *Look-Ahead Based SAT Solvers* `https://www.cs.utexas.edu/~marijn/publications/p01c05_lah.pdf`
- Huele, Marijn. *Guiding CDCL SAT Solvers by Lookaheads* `https://www.cs.utexas.edu/~marijn/publications/cube.pdf`
- Heule, Marijn. *Look-ahead SAT Solvers: Smart vs. Fast* `https://www.youtube.com/watch?v=cqPg-45KJp0`
- van der Tak, Peter. *Concurrent Cube-and-Conquer* `https://fmv.jku.at/papers/TakHeuleBiere-POS12.pdf`
- UBC Vancouver. *SATLIB Benchmark Problems* `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`
- Florida State University. *CNF Files* `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`
- Gelisam. *2-Dimensional Algebraic Data Type* `https://www.reddit.com/r/haskell/comments/1oj3x7/2dimensional_algebraic_data_type/`
- Marlow, Simon. *Parallel and Concurrent Programming in Haskell* `https://www.cs.utexas.edu/~marijn/publications/cube.pdf`
- Brecknell, Matthew. *Difference Lists* `https://matthew.brecknell.net/posts/difference-lists/`