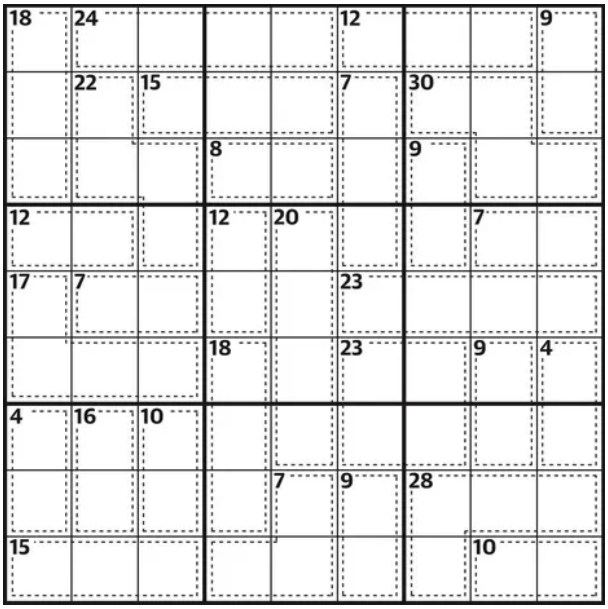
Parallel, Conflict-Driven DPLL SAT solver for Killer Sudoku

Liam Gersten | Alexander Wang | GitHub

SUMMARY

We plan to solve Killer Sudoku using the famous Boolean Satisfiability problem in parallel. Our parallel SAT solver implementation will build on DPLL, and will make use of conflict clauses.

BACKGROUND



A Killer Sudoku instance must be reduced to a conjunctive normal form SAT instance, which can be sequential or parallel if it is computationally intensive. From there, all threads cooperate with their private address spaces and message passing to run the DPLL algorithm with conflict resolution on the CNF formula. This problem stands to benefit from parallelism due to the massive search space, and an even larger clause number required.

To briefly explain DPLL, we pick a variable in our formula, and try setting it to true or false in recursive calls. When we assign a variable to a value, we replace every instance of it in the formula in what's called "unit propagation". This lets us simplify the formula, possibly eliminating clauses entirely. When a clause contains just one variable, we already know its assignment so we can propagate that immediately without recursing.

Conflict resolution involves backtracking up the call stack and adding a "conflict clause" to the formula before proceeding down again. We create a conflict clause to send back up if we find a clause cannot be satisfied, like (!x) when we have x assigned to true currently. It is intended to help future calls not make the same mistake just made. If we think of the call stack in a tree-like way, there are possibly more subtrees elsewhere that are identical, and we'd like them to avoid doing the same thing. The clause we send up is generated using the resolution algorithm on several other clauses.

CHALLENGE

In Killer Sudoku, an n * n board (n is a square) will need 9n^2 variables if we represent one variable for each of the nine values a single cell can take. There must be pairwise restrictions for these variables in the same rows, columns, chunks, and chunks with explicit summations. The number of clauses from such restrictions exceeds 7n^3. With integer representations, we start to run into overflow for n > 676.

Parallel or not, this problem can't be brute-forced. For just n = 9 as seen in the figure above, we have a search space of 2^729 variable combinations, far exceeding the particles in the known universe. With DPLL, we recursively divide the search space, and make shortcuts along the way. The algorithm works over an

cuts out parts of a formula which will need to be held in memory. The issue arises when one thread wants to try assigning variable x = True, and another tries x = False. These will entail different updates to the same memory in the formula. For this reason, it makes the most sense to give threads a private address space and use the message passing model.

One challenge with using private address spaces is that different values for the same variable may lead to drastically smaller or larger call stacks, so we have an inherent load balancing issue. We must then implement some form of work stealing, which is tricky with message passing, since a thread must grab the entire problem context (formula + assignments) at some stage in the tree. A very conservative estimate for the number of clauses needed is 27n^3, so if we have a call stack of up to 9n^2 calls, keeping a different formula version at each call (for stealers or backtracking) would require up to 243n^5 clauses saved at any given time. Even just storing pointers to clauses, we'd use up to 18gb of memory per thread at n = 25. The only way to cut this down is by editing the same (aliased) formula across the call stack, maintaining an efficient edit history, and using this history to reset parts of the formula when backtracking.

Creating and sending conflict clauses requires both backtracking up the tree, and having a linear history of decisions and unit propagations made down the call stack. It is unclear how a thread is supposed to backtrack out of stolen work to the correct call in the original thread. It’s also possible our program could benefit from telling other threads of a new conflict cause so they may avoid the same mistake made, although it is unclear if this should be anything other than a broadcast of a conflict clause. We’ll need to have threads periodically check for conflict clause messages, as well as ones that instruct the program to halt if a solution is found.

Backtracking becomes even more of an issue when a thread is done with its recursive call, but the other call (trying opposite value) has been stolen. The thread can't just return to the parent normally, as it needs the result of the stolen work. We don't want the thread to idle while waiting for the stealer's result, so it must either steal another thread's work in the meantime, or backtrack/return to do some of its own work while continuing to wait for the stealer's result.

Another problem comes with how the work division does not nicely divide with the private address space model. At the start of the call tree, we are working sequentially until we split by calling recursively. With private address spaces, we can’t just “spawn” other instances, they all start at once, so all threads will be doing the same thing until division. When trying variable assignment, a thread needs to know whether to try both, or try just one if another thread is going to be handling that work.

RESOURCES

As we will be using a message passing model with private address spaces, we would like to use the same compute resources available to us for assignment 4. While not strictly necessary, it would be helpful to have access to the PSC machines towards the end of our development cycle. Code-wise, we will be starting only with two header files created during assignment 4, those being the unimplemented state and interconnect classes. These act as abstract ways for threads to communicate and update their memory. Our understanding of the DPLL and conflict-driven SAT solvers is based largely on this 15-414 lecture, although it does not explicitly mention parallelism in the approaches.

GOALS AND DELIVERABLES

Our primary deliverable would be a message passing version of a parallel SAT solver, with an emphasis of optimization towards killer sudokus. This requires three sub-features: firstly, a program which can randomly

generate an arbitrary-sized killer sudoku puzzle; second, a way to convert such a puzzle into CNF; and thirdly, the SAT solver itself. During the research to select our final project, we found one brief resource online discussing a killer sudoku SAT solver, recording an average solve time of 0.21s per 9x9 using PycoSAT. As PycoSAT is a professional, open-source SAT solver in C, and we are writing our own from scratch - although ours would be parallel, a priori it seems reasonable to aim within a small factor of that time - say, under a second for n=2 or 4. Secondly, we should aim for a reasonable speedup graph, to keep our emphasis on cooperation between threads. In this aspect, based upon previous homeworks and some summary results online of state-of-the-art cooperative SAT solvers, a reasonable goal could be 2-3x speedup on n=8.

Accounting for unforseen spikes (or lack of) difficulty during our work, we may decide to pursue the sharing of conflict clauses with other threads, or simply keep that information local per thread. Additionally, adapting our solver to other sudoku variants is straightforward - and in the opposite direction, if killers are too difficult, we may drop back to normal sudokus as well.

A demo is doable for our project. We can show how specific SAT optimizations translate into a more human-oriented strategy of approaching the puzzle. We will also (hopefully) show a non-trivial speedup graph. Another interesting graph could be how our solver, on killer sudokus, compares to other widely-available public SAT solvers. Something cute could be - knowing the fact that sudoku is NP-complete - we can encode a simple instance of another NP-hard problem as a sudoku puzzle, then solve it.

PLATFORM CHOICE

The DPLL algorithm is based on reducing the search space by taking shortcuts. It does this by erasing (making constant) entire literals or clauses so they don’t need to be looked at again. Threads will need to be able to independently update the formula with their choices without being subject to the choices other threads are making. Considering almost no memory is shared between threads at all, separate address spaces make the most sense here. The GHC machines paired with C++ and MPI will work perfectly fine for our purposes, although our project could benefit from limited access to the PSC machines due to the problem size constraints.

SCHEDULE

Week 1. Complete a basic interface for SAT (literals, clauses, etc) as well as our SAT-specific message-passing interface.

Week 2. Implement a sequential SAT algorithm (involving DPLL and/or conflict clauses), as well as the sudoku generator and sudoku-to-CNF programs.

Week 3. Extend our implementation to correctly involve MPI.

Week 4. Debug MPI, plus theorize and implement several different strategies regarding work distribution and load balancing and work stealing.

Week 5. Optimize w.r.t sudoku strategies, message communications, CNF-encoding, etc.

Week 6. Final optimizations and assembly of final report.