Parallelizing a Sudoku Solver

Authors: Nicholas Grill (ngrill), Kevin Song (kevinson)

Webpage: https://njgrill.github.io/418-final-project/

Summary

We used OpenMP on the GHC machines and the PSC cluster to parallelize a brute force sequential Sudoku solver, and compared the performance against the fastest single-thread implementation. We achieved a 35x speedup with 32 cores on PSC over the best sequential solver on 25x25 Sudoku boards.

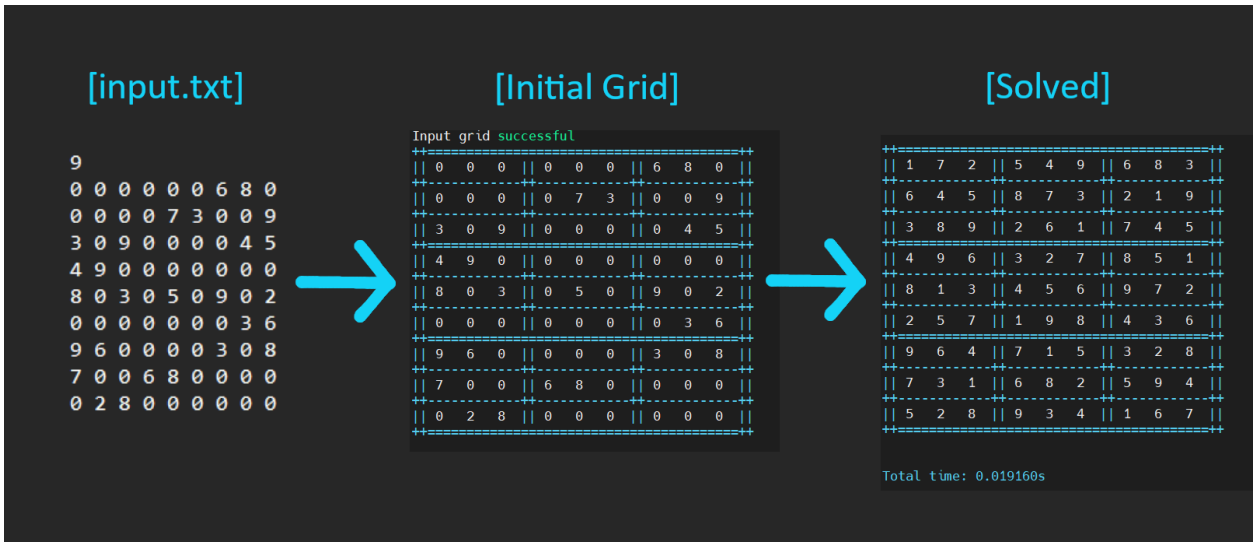
Background

We improved upon an existing brute force Sudoku solver by adding pattern-solving and parallel computation capabilities. Specifically, we added pattern-solving to improve the sequential work as much as possible, and then employed parallelism to speed up the brute force portion of the algorithm.

The brute force algorithm for solving Sudoku is fairly simple; it iterates through the empty cells of a Sudoku grid and, at each empty cell, computes which values would be valid considering the current values in the row, column, and subgrid. Using this list of possible values, the algorithm chooses one, writes it to the cell, then continues brute-forcing through the next cell. If it turns out that no solution was found this way, it chooses a different value from its possibilities list and tries again. Ultimately, this algorithm is just a specific instance of depth-first search on the Sudoku board.

The main data structure in this algorithm is the board, represented as a 2-D vector in C++. We also used a stack as well as a mutex lock in an early iteration of the algorithm, both of which will be discussed later in the "Approach" section.

We modified the code so that the user could specify a board in a .txt file and redirect that to be the input when running the program. It also accepts as input a filename to save the resulting, soled board to. At the end of the program, it also prints the final board as well as the time it took to solve it, as shown below:



[The above figure shows the input-to-output process of our program]

The most computationally expensive part of the algorithm is definitely the brute force section. This is due to the fact that, as with most brute force algorithms, the runtime is exponential in terms of the input. Specifically, for an $n \times n$ Sudoku board, the runtime can take on the order $O(n^{n^2})$ time, and general Sudoku solving is known to be NP-complete.

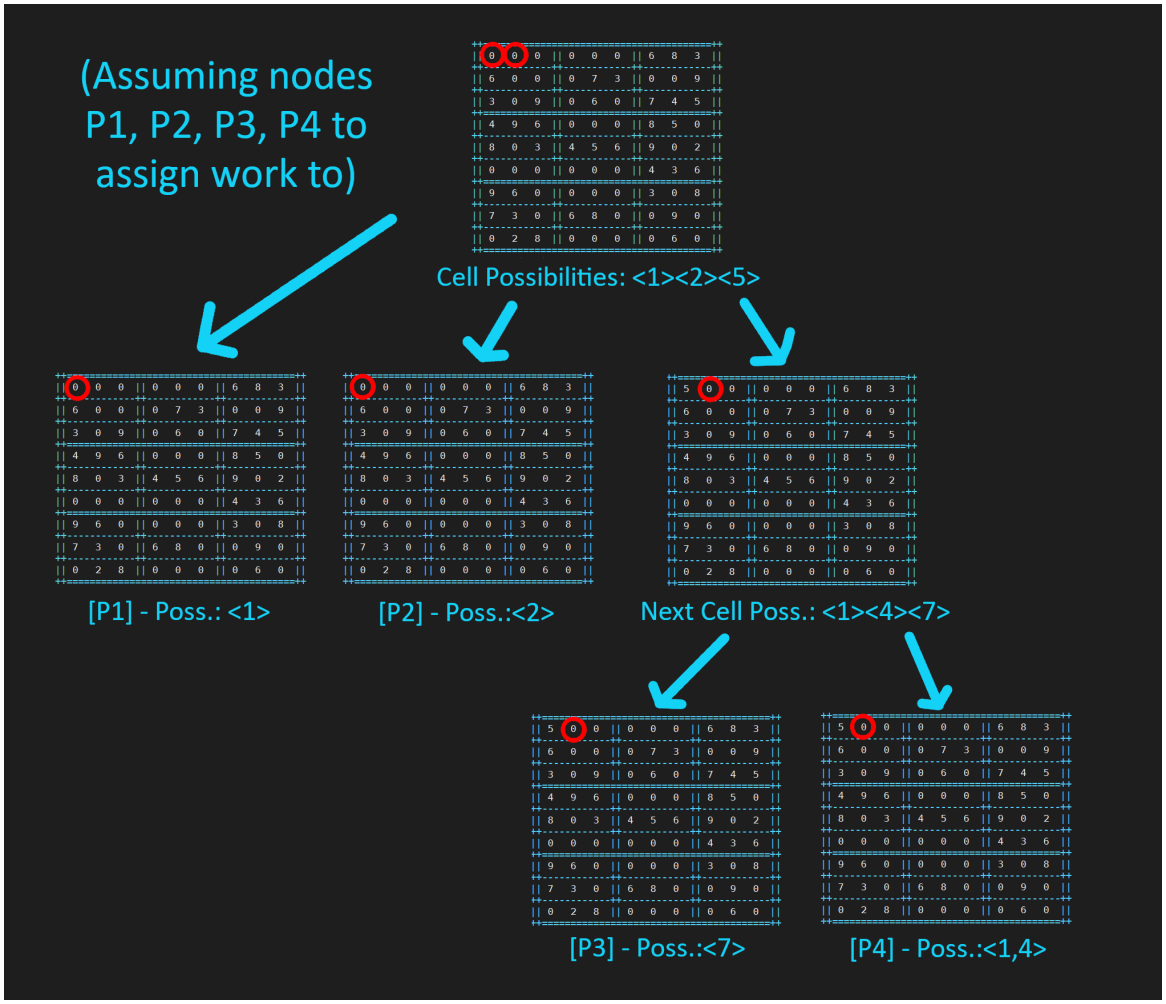
Depth-first search is, at its most basic level, not parallelizable. The runtime can also vary heavily between same-sized inputs as whether or not it goes down a correct path or one leading to a dead-end is not known from a current node. That said, the two parallel approaches implemented try to consider as many paths as possible in parallel to decrease the expected time taken. There is also some locality in the workload in terms of visiting new nodes, but the computation of possibilities for a given node does involve checking an entire row, column, and subgrid and does not benefit much from locality.

Approach

The core of our approach relies on using the OpenMP framework in C++ to run the brute force algorithm on multiple "board states" in parallel. The actual definition of a "board state" varies slightly between our two main approaches, but can generally be thought of as a Sudoku board where some blank cells are already filled in and act as though they are "uneditable." We initially targeted the GHC clusters and used them for all creating, debugging, and testing, but later ran performance tests on the PSC clusters to make use of the greater amount of cores.

We built off of a sequential brute force Sudoku solver found on GitHub here, but in the end almost none of our code is the same (except for the validator, used only as a means of testing our code). It should also be noted that between the time we downloaded their repo and the time we finished our code, they have pushed new commits; the link provided goes to the commit we pulled from.

We first split up the initial board into multiple, independent board states such that each OMP thread could run the brute force algorithm on its individual board. This is done in a breadth-first search manner until enough different boards have been created to occupy all threads; in the rare case that the board is solved by this BFS before populating every thread, we immediately return a correct result. An instance of this process on a sample board is shown below:



[The above figure illustrates the initial-grids initialization process]

If any thread reaches the end of its current board (i.e. finding a solution), it notifies the other threads that a solution has been found and writes its board to the "final_board" variable to be returned.

This communication with other threads, and some other communication, is handled by a global array of "shared_vals" structs which contains variables such as the thread's Sudoku grid, whether a solution has been found, and some other useful information. Importantly, although it is a global array, the structs themselves are padded to some multiple of a cache line to avoid false sharing.

The last important part of our approach (with the greatest impact on performance) was determining what to do when a thread exhausts its current board and finds no solution. We ultimately implemented two approaches, which we describe below.

Approach 1: Global Task Stack

Our first approach involved using a global task stack. For our purposes, a task was a specific "board state," meaning a Sudoku board with some values already filled in alongside a row and column index representing the first blank cell to start running the brute force from. The stack used was the C++ standard library stack.

When a thread needed a new grid, it gained access to the task stack by using OMP's mutex lock. After locking the stack, if the stack was nonempty then it would pop a board state off, unlock the stack, then run the brute force algorithm on this new board. If the stack was empty, it notified its neighboring thread (the thread with id 1 greater than its own). At that point, the neighboring thread would add more grids to the stack before computing its current cell. This is done in a similar manner to the initial grid population method done via BFS.

It became clear that this approach created a bottleneck for our performance in a number of ways. For instance, the number of stack lock accesses averaged at 343 for the "unsolved16x16.txt" board, with the average time taken to lock the stack, populate the stack with more grids, and unlock, as 0.000122 s. Given that the sequential code could solve the entire board in 0.000844 s, our options were either to reduce the number of lock accesses that occurred by over 99%, or reduce the time taken to populate the stack with more grids significantly. Our second approach attempts to tackle both of these bottlenecks.

Approach 2: Task Stealing

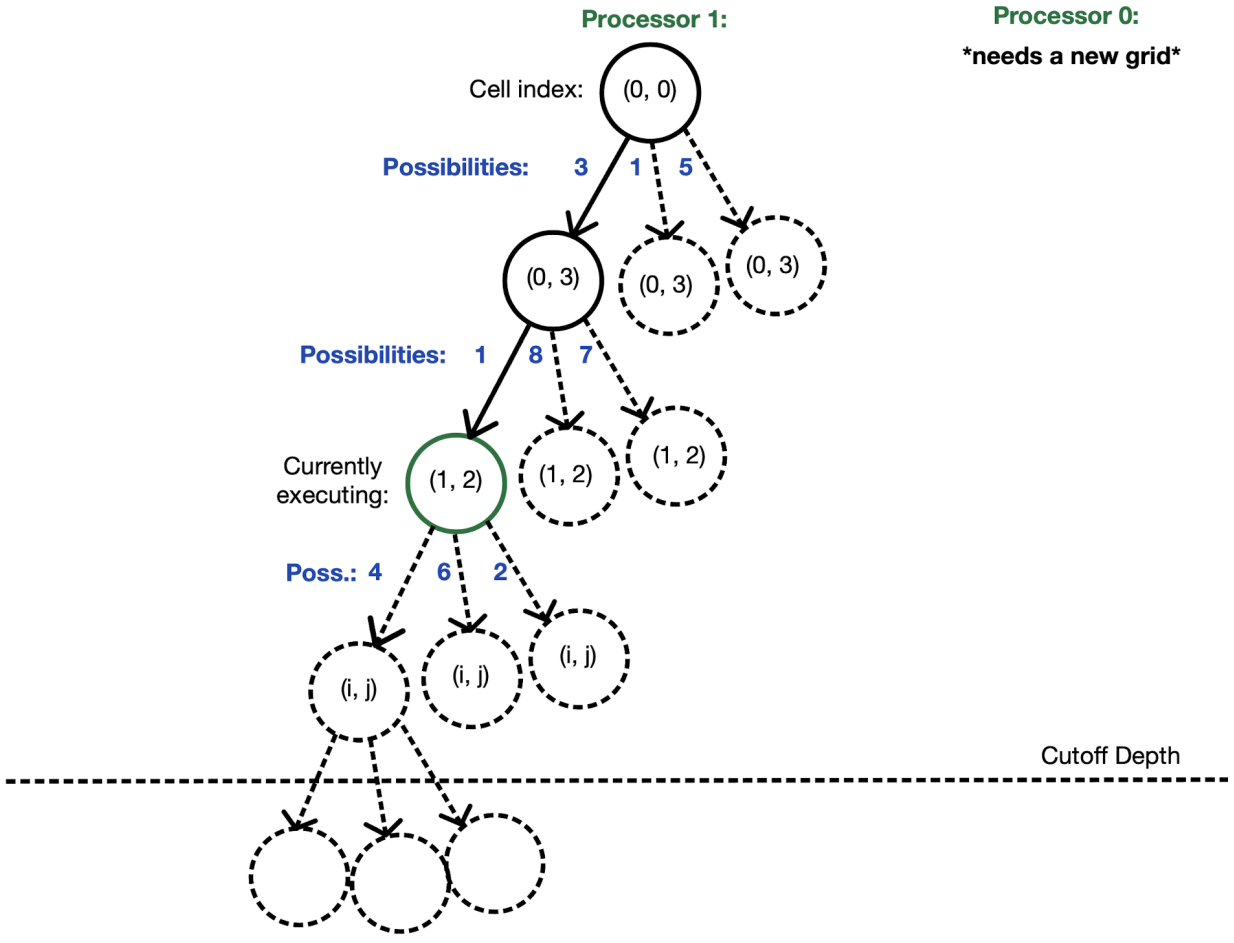
This approach is based on task stealing where, instead of relying on a global stack, a thread directly steals some work from another thread. This approach is entirely lock-free, removing the overhead from locking found in Approach 1. As we will show, it also reduces the time taken for a thread to receive a new board, since we can directly copy only the values we need to the requesting process.

To implement this, we now append our "board state" with a "possibilities grid" which, for each Sudoku cell, contains the possibilities available for that cell that have been computed thus far. Note that in the previous approach these possibilities were implicitly kept track of by for-looping over the possibilities for a given cell and, within the for-loop, recursively calling the solving

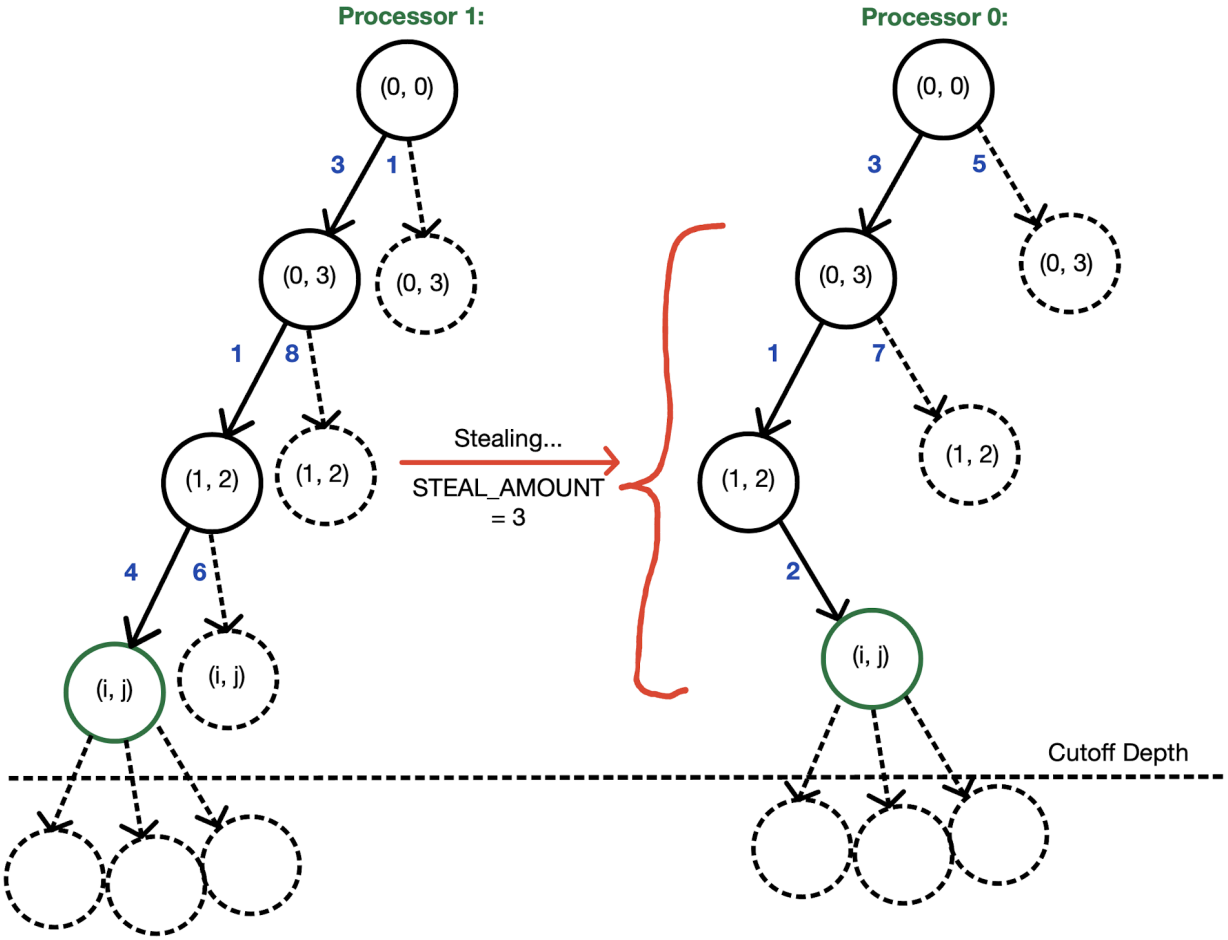
algorithm after having set that cell's value to the current possibility in the for-loop. This allows for more flexibility with regards to work granularity, as we will show below.

When a thread finds no solution on its board, it once again notifies its neighboring thread. This time, instead of adding to a global task stack, the neighboring thread copies over its own board, splitting up the possibilities at each node. Given the randomness involved in DFS, the best way to split possibilities is exactly evenly, but the number of nodes to split is tuned by a hyper-parameter STEAL_AMOUNT. We also introduce a cutoff depth for granularity purposes after which no stealing occurs. The following diagram shows the before and after of the stealing process on a sample board state:

Before:



After:



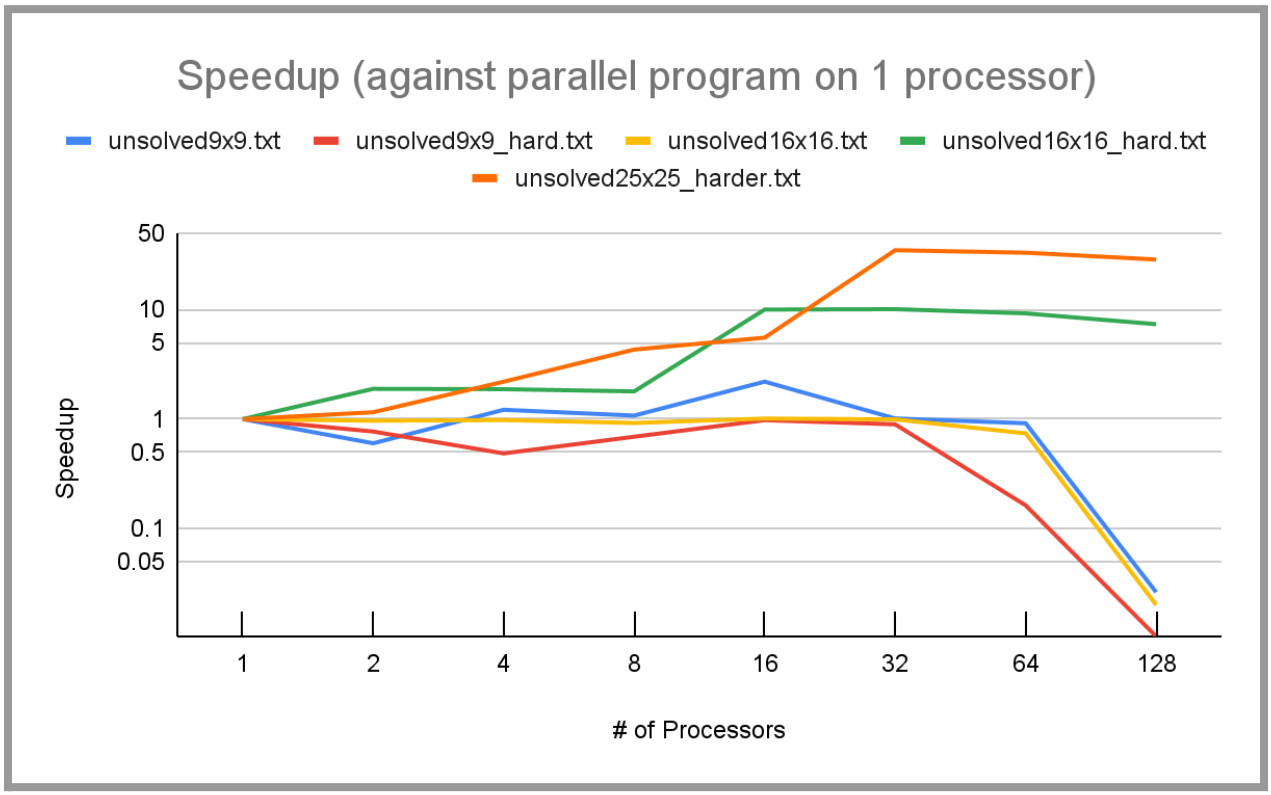
Further Optimizations:

- We altered the Sudoku grid from being an 2D int array to being a 2D vector of char-vectors. This decreased both board initialization and board stealing times.
- We initially had an additional 2D boolean array to mark which cells were “editable”, but we later wrapped the “editable” boolean property into the MSBs of the chars in the previously mentioned vector of vectors.

Results

Our primary performance metric was speedup in the time taken to solve a Sudoku board compared to the parallel code on one processor. To collect this data, we simply input the board, then placed timers around the "solve" function and averaged the time over multiple iterations (50 iterations for faster boards and 10 for slower ones).

The variance of speedup against the number of processors used on the Pittsburgh Supercomputing Cluster can be seen below:



[Plot 1: Speedup vs. # of Processors]

The above results were calculated using unlimited STOP_DEPTH and a STEAL_AMOUNT of 1. The best speedup achieved was 35x on the "unsolved25x25_harder.txt" board using 32 cores (with similar speedups for 64 and 128 cores).

It is clear from Plot 1 that our algorithm depends heavily on not only input size, but input difficulty. For the "unsolved9x9*.txt" boards, the solve time in the single thread version was too

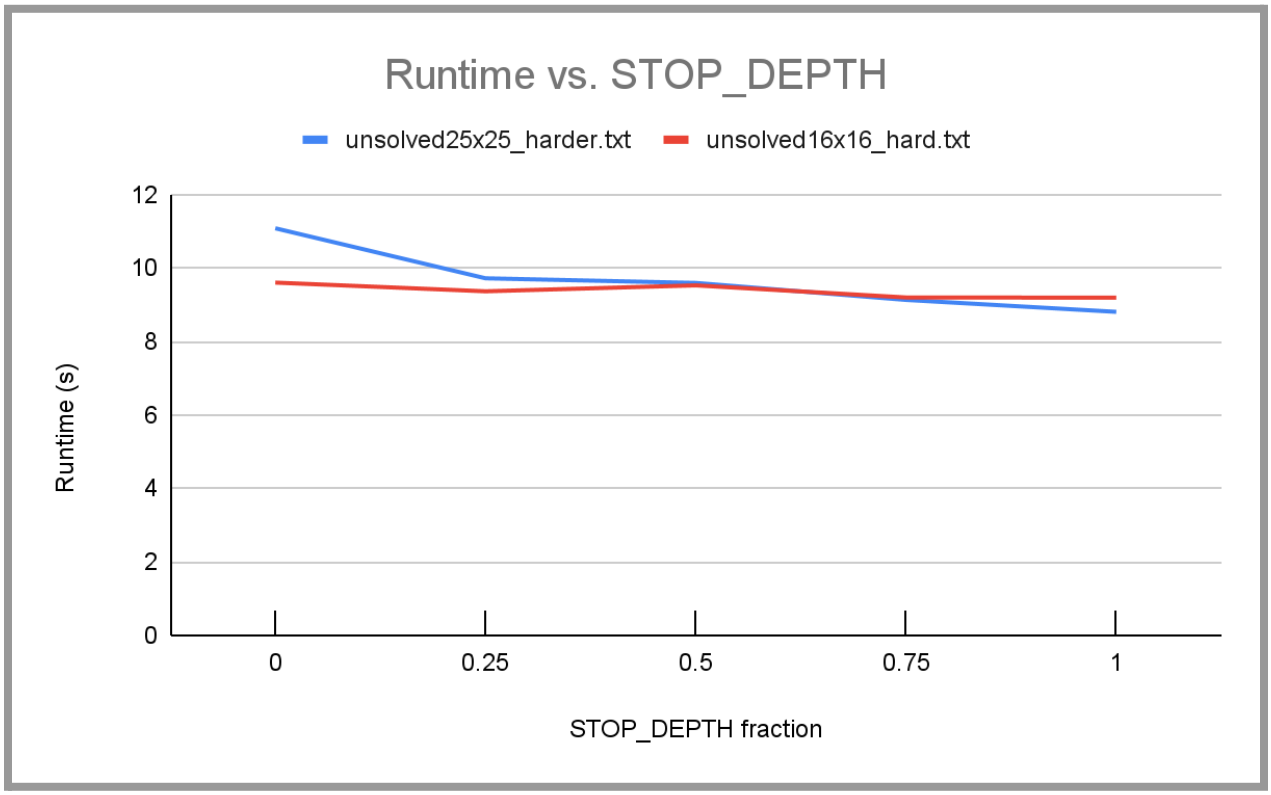
fast to mask the overhead of using multiple threads. For both “unsolved9x9.txt” and “unsolved9x9_hard.txt” the average runtimes were 0.000659 s and 0.000268 s, respectively. This is due to the fact that the sequential pattern solving portion of the code eliminates almost all of the board possibilities, taking up a vast majority of the runtime: the sequential portion of the code for these boards is 36% and 45%, respectively, as compared to between (2% and 15% for other board types). Amdahl’s law tells us that because the portion of runtime spent on sequential code is so high, we can’t achieve very good parallelism. In the best case for the two boards in question, we can get at most ~3x speedup; the best we get in these cases is a 2.2x speedup on 16 cores for “unsolved9x9.txt.”

The “unsolved16x16.txt” board also failed to achieve good parallelism, but not because of board size (“unsolved16x16_hard.txt” speeds up well). The reason for the slowdown in this instance is that there were multiple valid solutions. In this case, no stealing occurs because processors never encounter invalid solutions. As a result, the average runtime for a given thread will be similar to the single-thread version, and the overall runtime will slowdown due to the overhead of splitting among the threads. A more extreme example of this issue can be seen in Plot 3.

For the remaining “unsolved16x16_hard.txt” and “unsolved25x25_hard.txt” boards, we do see significant performance increases. This is due to the fact that the “*_hard.txt” suffix indicates a board with only one valid solution, most similar to a normal Sudoku puzzle. As such, the ability to search multiple paths in parallel now offers a significant performance boost, as the likelihood of finding the valid solution faster than the sequential approach increases linearly with the number of processors.

We also note how speedup increases with board size by examining the three “unsolved*x*_hard.txt” boards. As mentioned before, the sequential portion made up 45% of the runtime for the 9x9 board, but for both the 16x16 and 25x25 boards, the sequential portion only made up around 0.011%. This is a combination of the fact that pattern solving can’t shrink the search space down by the same proportion on larger boards, and the fact that the DFS paths are longer on larger boards, allowing more chances for stealing to occur.

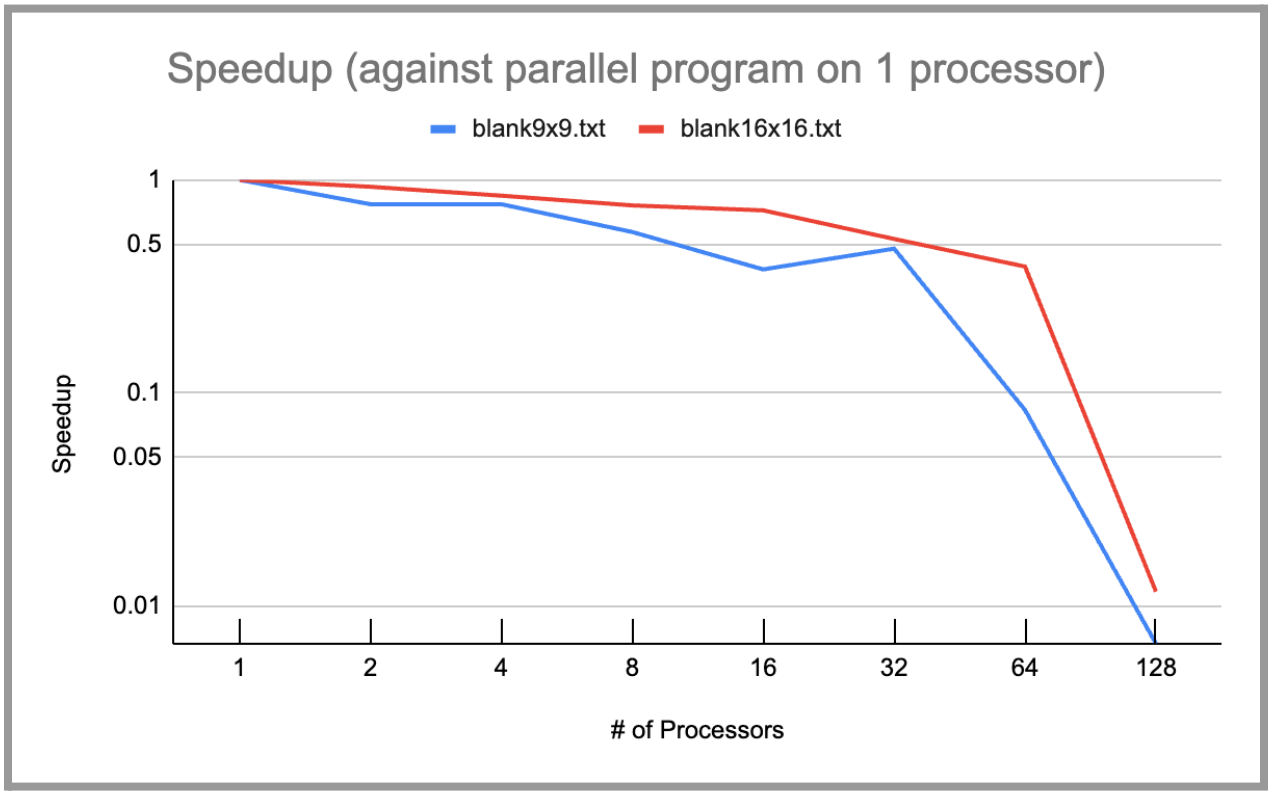
The following graph displays the speedup of our final solver implementation as the value of the stop-depth (as a fraction of the possible frame depth, where 0.0 indicates no sharing and 1.0 allows task stealing at any depth) is increased.



[Plot 2: Speedup vs. Stop Depth]

The above results were computed using 128 cores on the Pittsburgh Supercomputing Cluster and a STEAL_AMOUNT of 1. As can be seen, there is a correlation between STOP_DEPTH and overall runtime. In the "unsolved25x25_harder.txt" case, we see a significant decrease (20%) in runtime between STOP_DEPTH values of 0 and 1. In other words, the difference between disallowing stealing and allowing stealing all of the time lead to a 20% decrease in time for this board. We found the optimal STOP_DEPTH fraction for most boards was between 0.5 and 0.75. Unfortunately, on the GHC machines, we were unable to observe a statistically significant change in speedup, due to the pre-existing variance in solve times as well as the low number of threads. With 16 threads, the highest steal-count observed was 3 on "unsolved16x16_hard".

The following graph displays the "speedup" experienced when solving blank grids with different numbers of processors splitting the work of the task.



[Plot 3: Speedup vs. # of Processors on Blank Boards]

As demonstrated above, there is no case where having more than 1 processor working on the problem results in performance benefit; in fact, having more processors results only in performance decreases. This effect is caused by the nature of blank Sudoku grids, which do not have "dead-ends"/backtracking when applying a brute force algorithm. For instance, splitting a blank9x9.txt board among 9 processors by considering each value for the first cell offers no speedup over sequential brute force since all boards will lead to valid solutions in roughly the same amount of time. The only effect of parallelism on the solving of these grids is the addition of communication costs, resulting in only performance decreases.

Analysis of Results

During our generation of our final results data, we discovered that running one thread of our parallel implementation was much faster than running our sequential pattern-solving implementation. This difference in performance is most likely due to other algorithm changes we made later on (including refactors of the Sudoku frame and Sudoku solver classes). The performance impact of these changes were not evaluated (as they were not major algorithm changes), so the build-up of several of these changes could explain the observed performance differences. We also found that the parallel code with 1 processor did not introduce significant overhead, which is why many of our speedup results utilize the parallel code on 1 processor as a proxy for the "best" sequential algorithm.

Despite the lack of parallel resources, our sequential pattern-solving algorithm performed better than several of our initial parallel algorithm iterations. This is likely due to the fact that Sudoku as a problem often features a large search space, which means that algorithms that reduce that search space fare significantly better than those which don't. Since our parallel algorithms focused on evenly splitting solving work rather than reducing work, they were unable to produce significantly more speedup than our best (at the time) sequential implementation.

Overall, the performance gains of our combined solving implementation can be attributed to the fact that their optimizations are independent of each other; pattern solving reduces the search space early on, and parallel DFS tasks search through the remaining space much more quickly.

References

Code Base for Sudoku Solver:
https://github.com/ArjArav98/Sudoku-Solver/tree/41632183726e957aa33a2cb8ef0e870604b0e1d4

Parallel Depth First Search Inspiration (Approach 2):
https://www.lrde.epita.fr/~bleton/doc/parallel-depth-first-search.pdf

Common Sudoku Patterns: https://www.ijcsi.org/papers/IJCSI-11-2-1-247-253.pdf

Work Distribution

- Fixing/modifying original code base to work on larger grids (Nick/Kevin)
- Creating test cases (Nick/Kevin)
- Adding grid input/output (Nick)
- Pattern solving implementation (Kevin)
- Parallel Approach 1 design/implementation (Nick)
- Milestone Report (Kevin)
- Fixing/formatting grid output (Kevin)
- Parallel Approach 2 design/implementation (Nick)
- Adding timing metrics (Kevin)
- Parallel Approach 2 debugging (Nick/Kevin)
- Add STEAL_AMOUNT feature to Approach 2 (Nick)
- Decreasing/optimizing “board state” size (Kevin)
- Implement pattern solving before brute force in Approach 2 (Kevin)
- Hyperparameter tuning (Nick)
- Final report (Nick/Kevin)

Given the above, we believe the total credit should be split 50-50.