Parallelization and Large Scale Performance Testing of Sudoku Solvers

Ryan Stillings (661479855), stillr@rpi.edu

Abstract

Sudoku is a widely known logic puzzle, requiring players to place numbers in a partially filled 2d grid such that no row, column, or region contains any number more than once. Because solving a Sudoku puzzle is an NP-Complete problem[1] programmatically solving a complex Sudoko puzzle can be a very slow endeavor. In this project, I implemented two different approaches to solving Sudoku puzzles – brute force search and constraint propagation, and produced a serial and a parallel implementation for each. I then tested these algorithms on boards of varying complexities and cell removal thresholds at varying node counts, and graphed the results. Based on my evaluation, the brute force search algorithm performed poorly as the cell removal threshold increased, quickly becoming infeasible even when taking full advantage of many nodes, while the constraint propagation algorithm yielded a much better baseline, but did not scale quite as well when parallelized due to increased overhead introduced by message passing. I believe further improvements to my parallel constraint propagation implementation may be achieved by reducing the amount of time nodes spend polling for work request messages, as well as by the introduction of more thorough logic for initially divvying up work, in order to minimize the amount of time each node spends outside of running the constraint propagation algorithm itself.

1. Introduction

In this report, I present my approach at parallelizing two different algorithms for solving Sudoko boards using MPI on the Blue Gene Q supercomputer. The first section of this report details the motivation behind my work, as well as the contributions that I made. Section 2 gives a brief overview of the rules of Sudoku. Section 3 describes the details behind my project setup and implementation. Section 4 describes my experimental setup. Section 5 describes the results of my experiments. Section 6 provides some analysis into the results that my experiments yielded. Section 7 provides a review of related work. Section 8 describes my plans for future work. Finally, section 9 summarizes my findings and concludes this paper.

1.1. Motivation

This project is motivated by the popularity and range of difficulty inherent in logic puzzle games. Sudoku, along with its variants such as killer Sudoku[2] and hyper Sudoku, and similar, more recent logic puzzle games such as Str8ts[3], is a ubiquitous game with simple, easy to understand rules. As a result, research into games of this nature requires little explanation of the project domain, and can easily be related back to the game's simple rules and structure. At the same time, due to Sudoku's NP-Complete nature, fast, efficient solving requires a strong searching component, which has the potential to be parallelized in innovative ways. By parallelizing a Sudoku solver, it becomes possible to solve existing puzzles more quickly, while also making solving previously infeasible larger puzzles possible.

1.2. My Contributions

For this project I was the only member of my team. As such, all components of the project were contributed by me. This includes the initial research, the board generator, utility functions, all board solvers, the board validator, and the project write up.

2. Sudoku Overview

Although Sudoku is among the most widely known logic puzzle games, a brief overview will nonetheless be provided to ensure proper understanding before diving into discussion of the algorithms. Sudoku is a logic puzzle game which features an NxN grid of cells. While the dimensions of a Sudoku grid may be any square number, the most common grid sizes are 9x9 and 16x16, as any smaller grids would be trivial for humans to solve and any larger grids would be very difficult for humans to solve. In Sudoku, an initial grid is provided, with each cell containing either one or zero numbers. The goal of Sudoku is to place a single number in each cell, such that the entire grid is filled with the numbers between 1 and the dimension of the Sudoku board. Sudoku's rules additionally require that no row, column, or region contain the same number twice, meaning in a solved board each region will contain exactly one instance of each candidate number.

3. Implementation Details

My code base consists of a number of different components that directly and indirectly contribute to my solver implementations, each of which will be discussed in its own sections. The major components are board generation / loading, board validation, peer list population, the serial brute force solver, the parallel brute force solver, the serial constraint propagation solver, and the parallel constraint propagation solver.

3.1. Board Generation

The first major component of my project is board generation. In order to get an accurate measure of my solvers' performance, I needed the freedom to be able to automatically generate novel boards of any size and complexity. The first step taken by the board generator is creating a solved board using a template. Rather than manually supplying a template for each potential board size, I use a simple algorithm[4] which can be adapted to any board size using the following steps:

1. Fill in the first row with the candidate numbers in ascending order.
2. Fill in subsequent rows in ascending order, with the first number in each row offset by region size * rowNumber, wrapping back to 1 upon reaching the maximum candidate number.
3. Shift the values within each region to the right regionSize times, wrapping around to the left side, such that no number is repeated in any row or column.

At the conclusion of this algorithm we have our solved template board, adapting to the current board size, as shown in figure 1.

```
[ 1| 2| 3| 4] [ 5| 6| 7| 8] [ 9|10|11|12] [13|14|15|16]
[ 5| 6| 7| 8] [ 9|10|11|12] [13|14|15|16] [ 1| 2| 3| 4]
[ 9|10|11|12] [13|14|15|16] [ 1| 2| 3| 4] [ 5| 6| 7| 8]
[13|14|15|16] [ 1| 2| 3| 4] [ 5| 6| 7| 8] [ 9|10|11|12]
-------------------------------------------------------
[ 2| 3| 4| 1] [ 6| 7| 8| 5] [10|11|12| 9] [14|15|16|13]
[ 6| 7| 8| 5] [10|11|12| 9] [14|15|16|13] [ 2| 3| 4| 1]
[10|11|12| 9] [14|15|16|13] [ 2| 3| 4| 1] [ 6| 7| 8| 5]
[14|15|16|13] [ 2| 3| 4| 1] [ 6| 7| 8| 5] [10|11|12| 9]
-------------------------------------------------------
[ 3| 4| 1| 2] [ 7| 8| 5| 6] [11|12| 9|10] [15|16|13|14]
[ 7| 8| 5| 6] [11|12| 9|10] [15|16|13|14] [ 3| 4| 1| 2]
[11|12| 9|10] [15|16|13|14] [ 3| 4| 1| 2] [ 7| 8| 5| 6]
[15|16|13|14] [ 3| 4| 1| 2] [ 7| 8| 5| 6] [11|12| 9|10]
-------------------------------------------------------
[ 4| 1| 2| 3] [ 8| 5| 6| 7] [12| 9|10|11] [16|13|14|15]
[ 8| 5| 6| 7] [12| 9|10|11] [16|13|14|15] [ 4| 1| 2| 3]
[12| 9|10|11] [16|13|14|15] [ 4| 1| 2| 3] [ 8| 5| 6| 7]
[16|13|14|15] [ 4| 1| 2| 3] [ 8| 5| 6| 7] [12| 9|10|11]
```

Figure 1: Generated 16x16 template board

Once the template board has been generated, the generator performs a number of transformations on the board randomly, in order to produce a novel board. These transformations[5] are described below:

1. Randomly swap all occurrences of one number with another one. This allows us to break up the pattern within rows and columns produced by our template board.
2. Randomly swap rows within regions. This allows us to break up the row-level patterns produced by our template board, without violating the correctness of our generated board.
3. Randomly swap columns within regions. This allows us to break up the column-level patterns produced by our template board,

without violating the correctness of our generated board.

4. Randomly swap regionSize row groups. This allows us to break up the regionSize row-level patterns produced by our template board, while still preserving our board's validity.
5. Randomly swap regionSize column groups. This allows us to break up the regionSize column-level patterns produced by our template board, while still preserving our board's validity.

Once all 5 of these steps have been performed a sufficient number of times, our generated board will be completely free of any cell, row, column, or region-level patterns introduced by our template generator.

After our solved board has been sufficiently randomized, the last thing our generator needs to do is remove cells to produce a proper Sudoku puzzle that we can pass off to our solvers. There are two approaches we can take here; either we produce evil[6] puzzles (ones with only one solution) or we produce standard puzzles (ones with potentially multiple solutions). As considerable research has gone into applying these algorithms specifically to evil boards[7] I felt I would gain more insight by lifting the single solution constraint, and seeing how these algorithms perform on boards that can have multiple unique solutions instead. To this end I introduced a threshold value, dictating the percentage of cells to remove from our generated board. Once this number of cells has been removed, our board is complete, valid, and ready for solving.

Our board generator also includes support for loading the board directly from a file. As we wish to ensure correctness during the experiment phase, it is necessary that the set of experiments for a given board size and threshold all use the same board, which is where file loading comes in. Initial unsolved board output may be copied and pasted as input to the included helper python file if desired, which will produce a board file that may be loaded in during any subsequent run. The board file simply consists of the initial cell values ordered by row first then column, with each cell value separated by a single space. Unfilled cells are represented by the number 0, which is otherwise unused regardless of board size. Loading the board is then a simple matter of repeated calls to fscanf, populating an empty board in row then column order.

3.2. Board Validation

Our board validator iterates over each cell in the board, checking if any cell violates any of the rules of Sudoku. The validation logic is straightforward. First, the validator check's the current cell's row and column, to determine if any other cell in either that cell's row or that cell's column contains the same value as that cell. If so, the board is deemed invalid. Next, the validator iterates over each region, producing an ordered list of all of the values present in that region. If any value is absent from that region, the board is deemed invalid. If these checks pass for all cells and all regions, the board is deemed valid and solved. Faster related methods also exist for the convenience of the solvers, including checking the validity of a single cell, and checking if the entire board is filled ignoring board validity.

3.3. Peer List Population

A key part of most Sudoku solvers involves the concept of peers[8]. A peer is considered to be any cell which shares the current cell's row, column, or region. Each cell has exactly 2*(boardSize-1) + regionSize*regionSize - 2*(regionSize-1) – 1 peers[9], as demonstrated in figure 2 below.

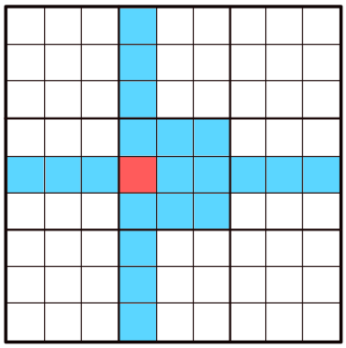

Figure 2: Peer list for cell at row 5 column 4

Because each cell's peers depend only on the size of the board, the list of peers can be pre-calculated and provided to the solvers with no additional cost. As a result, as part of my project setup I provided my solvers with a full list of peers for every cell on the board, which may be indexed into by cell row,column. The peers themselves are obtained using the same logic as described in the validator, iterating first through the current cell's row and column, and then finally through its region, being careful to avoid duplicates in the process.

3.4. Serial Brute Force Solver

The first solver I wrote for this project is a serial brute force solver. This is the simplest of the four algorithms, designed to serve as a baseline for performance and scalability comparisons. The brute force solver is unique in that it spends almost all of its time in the search phase[10], stopping at each iteration only to check if the board is filled, and if so, whether or not it's solved. Until the board is filled up, this solver selects the next unfilled cell, and recurses on all potential values for that cell. Because this solver tries every possible value in every cell, it is guaranteed to find a solution eventually. Due to its simple logic and lack of optimization, it is also easily parallelized, serving as a benchmark to compare parallelized speedup in a simple algorithm without much rank communication, to parallelized speedup in a more complex algorithm with frequent message passing. However, as it is inherently unoptimized, the brute force approach is likely to spend a large amount of time in the search phase, as at worst it must try every single combination of values in the blank cells. Due to how little logic the serial brute force solver performs outside of the search phase, it can be expected to perform quickly on small boards with few missing cell values to fill in, but scale very poorly[11] as more and more missing cells are introduced.

3.5. Parallel Brute Force Solver

The second solver I wrote for this project is a parallelized implementation of the brute force solver. This solver follows similar logic to the serial brute force solver, recursively trying values in the search phase until achieving a filled board, and returning that board if it is valid or popping the recursive stack and continuing if it is invalid. Where the parallelized brute force solver differs, however, is in selecting the values to recurse on during the search phase. As we now have the benefit of some number of ranks operating in parallel, we need to distribute work as evenly as we can across those ranks[12], rather than just having every rank test the same state tree branches. Because the brute force approach tries every possible combination of values for missing cells during the search phase without any constraints, we can divide up the search space effectively by leveraging each rank's rankNum. Our work division algorithm operates as follows:

1. Build a list of all valid candidates for the current empty cell.
2. If the candidate list is greater than or equal to the number of unaccounted for ranks, divide the ranks evenly among the candidate values and begin the search phase.
3. Otherwise, reduce the number of unaccounted for ranks by the number of candidate values, and recurse each rank on the candidate value corresponding to its rankNum

Using this algorithm the ranks will split up the search space as evenly as possible, potentially allowing for near linear speedup as more ranks are introduced.

3.6. Constraint Propagation

Constraint Propagation is a means of eliminating candidate values[14] from cells, which forms the foundation of the constraint propagation solvers. Candidate values may be eliminated via the following two rules:

1. If any peer has decided on a value, remove that value from the current cell's candidate values list
2. If all peers have eliminated a value from their candidate values lists, set the missing value as the current cell's only candidate value

By repeatedly applying these rules, the number of candidate values for each cell can be greatly reduced, shrinking the amount of branches that must be taken during the search phase.

3.7. Serial Constraint Propagation Solver

The serial constraint propagation solver forms the backbone of my project. This is a much more complex solver than the brute force implementation, using a combination of constraint propagation[13] and search on an array of cell value candidates. The serial constraint propagation solver begins by building up a grid of candidate cell value arrays. For given cells, the candidate list consists only of the given value, indicating that that cell is solved. For blank cells, the candidate list initially contains all numbers from 1 to the board size. Once our candidate value grid is initialized, we move into the recursive internal constraint propagation method. Here we run constraint propagation on all cells in row,column order, removing as many candidate values as we can. Any time we create a new singleton[15] (a cell with only one remaining candidate value), we perform another round of constraint propagation.

Eventually, we either fail to create a new singleton via constraint propagation, or we end up with a board where each cell has <= 1 remaining possibilities. If the latter is true, we check if the board is solved. If it is, we've found a solution. Otherwise, we've found a contradiction and much try recursing down another branch in the search phase. If we did not stop due to a full board, we need to transition to the search phase. Rather than simply searching the first blank cell we can find, we instead choose the cell with the fewest remaining possibilities, in order to maximize our chance of choosing the correct search branch.

Eventually we will return a correct value candidates grid with exactly one candidate per cell. At this point, we simply copy the possibilities to our board, and return the result. Due to all the work described above, the serial constraint propagation solver has to spend considerably more time outside of the search phase than the brute force solver does. However, because constraint propagation massively cuts down on the number of times we need to enter the search phase at all, this algorithm can be expected to perform far better on nontrivial boards than the brute force solver can.

3.8. Parallel Constraint Propagation Solver

The final solver implemented as part of my project is the parallelized version of the constraint propagation solver. Once again, this solver follows similar logic to my serial constraint propagation solver, with the key difference being in how work is distributed. All ranks start from the same point, running constraint propagation until it is time to search. When a rank must transition to the search phase, it first checks its list of already searched subtrees. If the desired subtree matches an already searched subtree, the rank skips over that subtree. Otherwise, it sends the new subtree to all other ranks, then continues its search in that subtree. Through this constant communication, ranks are able to ensure that they do not waste time searching an already searched subtree, which should result in a decent speedup as long as the overhead from checking and maintaining the searched subtree list isn't too great.

4. Evaluation

My evaluation was performed on RPI's Blue Gene/Q supercomputer. Tests were split into 4 categories: 9x9 50% cell removal threshold,

16x16 45% threshold, 16x16 50% threshold, and 16x16 55% threshold. For each test, the two serial algorithms ran once, and the two parallel algorithms ran on 1, 4, 16, 64, and 128 node configurations. The puzzles for these tests were generated randomly using the board generator described in section 3.1. Only a single cell removal threshold was tested for the 9x9 boards, as all solvers proved too quick to obtain meaningful data on the 9x9 boards.

5. Results

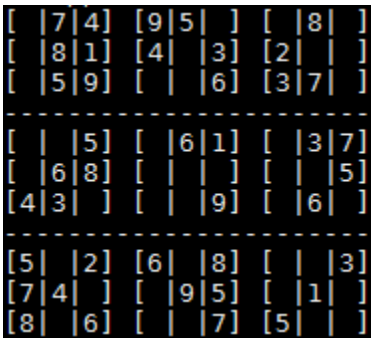

Figure 3: 9x9 50% threshold board

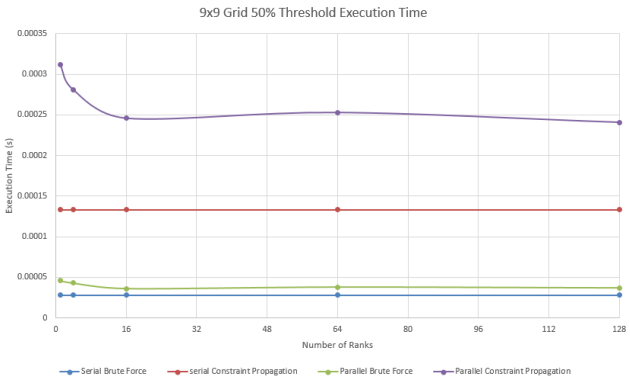

Figure 4: 9x9 50% threshold execution time

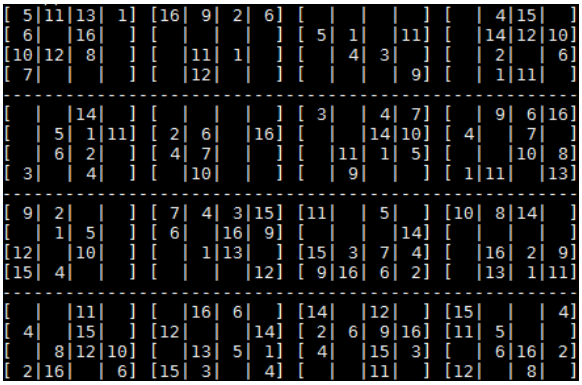

Figure 5: 16x16 45% threshold board

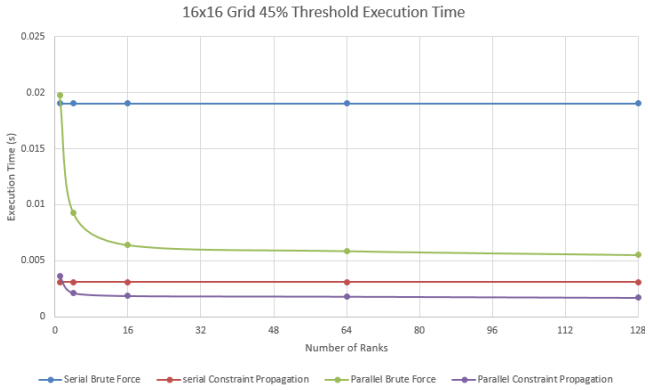

Figure 6: 16x16 45% threshold execution time

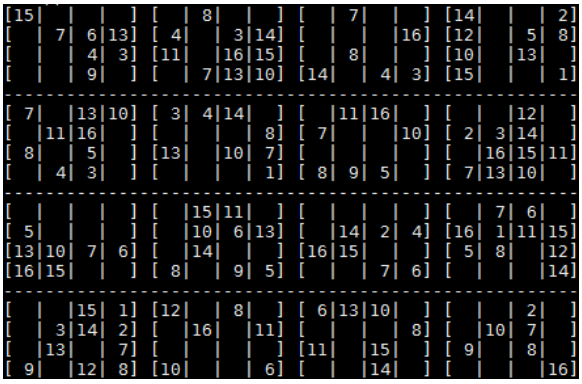

Figure 7: 16x16 50% threshold board

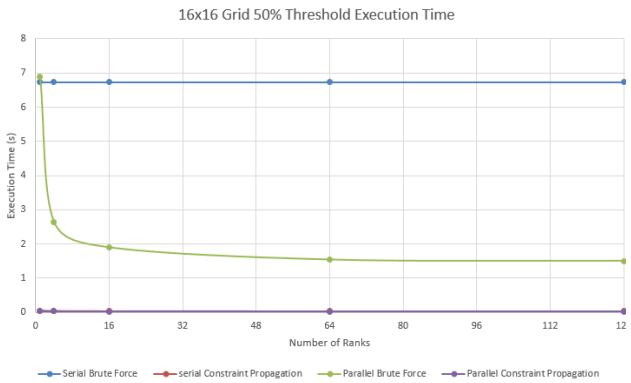

Figure 8: 16x16 50% threshold execution time

```
[   | 4|   | 6] [ 3|   |   |   ] [ 5|   |15|13] [ 2|   |11|   ]
[   |14|   |   ] [   |   |13| 5] [   |   |   |   ] [   |   |10| 6]
[   |   |   |   ] [   |   |   | 2] [   |   |   | 6] [ 5| 7|15|13]
[   |   |   |   ] [ 4|10|   |   ] [ 2|   |   | 9] [12|   |   | 8]
-----------------------------------------------------------------
[ 2|   |14|   ] [   | 5|   |   ] [ 3|   |   |   ] [   |   |16|   ]
[   |   |   | 1] [   | 2|11|   ] [ 4| 6|16|10] [ 7|   | 5|15]
[   |   |   |   ] [   |   |   | 4] [   | 9|   |   ] [ 3| 8|   |   ]
[16|   |   |   ] [   |12| 1| 3] [   |13|   |   ] [   |   |   |   ]
-----------------------------------------------------------------
[   | 2|   |14] [ 5|13| 7|15] [   |12| 8|   ] [10|16| 6|   ]
[   |16|10| 4] [   | 8|   |   ] [15| 5|13| 7] [11|   |   |   ]
[ 8|   | 1|   ] [   | 9|   |11] [10|   |   | 4] [   | 5|   | 7]
[   | 5|15|   ] [   |   |   |10] [   | 2| 9|   ] [ 1|   |   |   ]
-----------------------------------------------------------------
[   |10|   |16] [   | 3|12|   ] [13|15|   |   ] [   |11|14| 2]
[ 7|15|   | 5] [10| 4|16| 6] [ 9|   |   | 2] [   | 1| 3|   ]
[   |   |   | 2] [15| 7|   |   ] [ 8|   |   |   ] [   |10| 4|   ]
[   | 1|   |   ] [   |   |   | 9] [   |10| 4|16] [   |15| 7|   ]
```

Figure 9: 16x16 55% threshold board

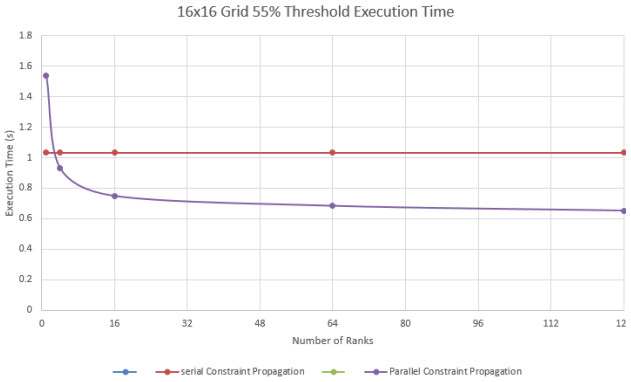

Figure 10: 16x16 55% threshold execution time

6. Analysis

Starting with the 9x9 50% threshold experiment, as shown in figure 4, the brute force solvers actually performed slightly better than the constraint propagation solvers, with both parallel solvers performing worse than their serial counterparts. This isn't too surprising; these puzzles consist of a very small search space, allowing brute force to run through the recursion tree and find the solution while constraint propagation is allocating the possibilities table and running through constraint logic. Consequently, the parallelized constraint propagation solver actually ran about 1x slower than the serial constraint propagation solver. Additionally, both parallel implementations saw only a very minor speedup as more ranks were introduced. This too was expected, as once again the search space is so small that very little search time can be reduced.

Moving on to the 16x16 45% threshold experiment, as shown in figure 6, we can start to see the impact of a much larger search space coming into play. Here brute force performed much worse than constraint propagation, with parallel brute force showing a sizable speedup over brute force, but still falling short of either constraint propagation solver. As for constraint propagation, the parallel solver with just a single rank performed slightly worse than the serial solver, which is to be expected as the parallel solver has some additional overhead even without the presence of other ranks. However, as the parallel constraint propagation solver was granted 4 ranks, it showed roughly a 50% speedup over the serial constraint propagation solver. The introduction of additional ranks only had a marginal benefit, suggesting that for this search space size the added complexity was approaching the benefit from increased searching speed.

As for the 16x16 50% threshold experiment, as shown in figure 8, the trends from the 45% experiment became even more pronounced. At this point the serial brute force solver took over 6 seconds to complete, with the parallel brute force solver trimming that down to around 1 and a half seconds at 128 nodes. The constraint propagation solvers also showed a similar pattern to that of the 45% threshold, with the parallel solver reaching a maximum execution time of roughly half that of the serial solver.

Finally, things became even more pronounced with the 16x16 55% threshold experiment, as shown in figure 10. Here, both brute force solvers failed to find a solution in the allotted time, suggesting that the search space had simple grown too large for

them to handle. This allowed us an opportunity to get a clearer look at the relationship between the serial and parallel constraint propagation solvers. In this final test, the parallel constraint propagation solver showed more pronounced scaling relative to the serial solver than in previous tests, most likely due to it being better able to take advantage of the increased search space. Overall, these tests demonstrated positive results, but suggest that a reduction in parallelization overhead would be necessary to achieve better scaling.

7. Related Work Review

Dan Taylor[16] performs an approach that is similar to my brute force solver, focusing on performing a depth first search through the search space, using recursive backtracking whenever a contradiction is reached. Bill DuPree[17] performs an approach that is similar to my serial constraint propagation solver, while adding on the naked/hidden subset and chute alignment constraints. Bill's solution does a stronger job of initial constraint propagation, but does not continue propagating constraints after switching to the search phase for the first time, resulting in poor scaling performance in larger puzzles with many unknowns. Russ Klein performs a thread heavy parallelization approach, with the master thread spawning a new child thread to traverse each search phase branch, and combing the child threads upon completion of their search. This implementation has the potential to scale very well when combined with proper constraint propagation, but is ill suited to MPI where the number of ranks is fixed at the start of the run. Finally, Alton Chiu et al.[7] analyze several parallelized constraint propagation approaches. Their approach is similar to my own, but uses threads with locks and message based work sharing rather than separate nodes and broadcasting explored subtrees. Further, they only run performance tests on up to 4 threads on a consumer grade machine, as opposed to testing large numbers of ranks on BlueGene/Q.

8. Future Work

For the parallelized constraint propagation solver, there are a number of improvements that would be worth exploring in the future. The glaring weakness of this solver is the searched subtrees array. While fine for small search spaces, it causes unbounded memory growth as the search space increases in size due to the lack of a deletion condition, and requires a worst case O(n) lookup time as each stored subtree has to be checked for equivalency before entering into the search phase. With regards to the lookup time, a possible solution would be to use a hashmap rather than an array. This would replace the linearly growing lookup time with a fixed hashing time, resulting in much better runtime scaling. Regarding the memory size, this is a much trickier matter. One minor but immediate improvement would be to only store the possibilities array for non-given cells. This would reduce memory usage by a decent amount, but would be a fair amount of work for just a moderate improvement. A more complete solution would be to not store searched subtrees at all, but rather to deterministically give ranks starting subtrees, and have them ask each other for additional work when idle. This would completely solve the issue of memory growth, but would introduce much more complexity to the algorithm, and potentially a performance penalty due to the increased amount of message passing overhead required.

9. Conclusion

In summary, this project introduces two Sudoku board solving algorithms, brute force and constraint propagation, each of which is implemented both serially and in parallel. This project also provides a strong foundation for future experiments, with a fast built-in board editor + loader, validator, and peer list populator. During the experiments, increasing the number of ranks used showed an immediate speedup with the parallel solvers on most puzzle boards. Parallelization typically netted a speedup of around 50-100% relative to the equivalent serial solver, mostly due to the associated increase in structural complexity and message passing overhead.

References

1. *"NP complete – Sudoku" (PDF). Imai.is.su-tokyo.ac.jp. Retrieved April 20, 2019.*
2. Stuart, A. "Sudoku creation and grading." Mathematica 39.6 (2007): 126-142.
3. Jeff Widderich “Home of Str8ts.” www.str8ts.com . Retrieved April 20, 2019
4. StackOverflow. https://stackoverflow.com/questions/6924216/how-to-generate-sudoku-boards-with-unique-solutions. Retrieved April 18, 2019.
5. DryIcons, “A simple algorithm for generating sudoku puzzles.” [Online]. Available: http://dryicons.com/blog/2009/08/14/asimple-algorithm-for-generating-sudoku-puzzles/
6. XUE, Yuan-hai, et al. "Sudoku puzzles generating: From easy to evil." Mathematics in practice and theory 21.000 (2009).
7. Chiu, Alton, Ehsan Nasiri, and Rafat Rashid. "Parallelization of sudoku." University of Toronto December 20 (2012).
8. Lloyd, Huw, and Martyn Amos. "Solving Sudoku with Ant Colony Optimisation." arXiv preprint arXiv:1805.03545 (2018).
9. Nguyen, Tuan T., Sang TT Nguyen, and Luu C. Nguyen. "Learning to Solve Sudoku Problems with Computer Vision Aided Approaches." *Information and Decision Sciences*. Springer, Singapore, 2018. 539-548.
10. Felgenhauer, Bertram, and Frazer Jarvis. "Mathematics of sudoku I." Mathematical Spectrum 39.1 (2006): 15-22.
11. Nicolau, Miguel, and Conor Ryan. "Solving sudoku with the gAuGE system." European Conference on Genetic Programming. Springer, Berlin, Heidelberg, 2006.
12. Saxena, Rahul, Monika Jain, and Syed Mohammad Yaqub. "Sudoku Game Solving Approach Through Parallel Processing." *Proceedings of the Second International Conference on Computational Intelligence and Informatics*. Springer, Singapore, 2018.
13. Simonis, Helmut. "Sudoku as a constraint problem." CP Workshop on modeling and reformulating Constraint Satisfaction Problems. Vol. 12. 2005.
14. Bartlett, Andrew, et al. "An integer programming model for the Sudoku problem." Journal of Online Mathematics and its Applications 8.1 (2008).
15. Crook, J. F. "A pencil-and-paper algorithm for solving Sudoku puzzles." *Notices of the AMS* 56.4 (2009): 460-468.
16. D. Taylor, “Solving every sudoku puzzle,” Logical Genetics. [Online]. Available: http://logicalgenetics.com/showarticle.php?topic id=1624
17. B. DuPree, “A sudoku solver in c.” [Online]. Available: http://www.techfinesse.com/game/sudoku solver.php
18. R. Klein, “Dude, wheres my multicore performance?” [Online]. Available: http://blogs.mentor.com/russklein/blog/2011/ 01/10/dude-wheres-my-multi-core-performance/