Multi-Threaded Sudoku Solver

Preprint · February 2020

DOI: 10.13140/RG.2.2.31103.36009

Ebin Scaria
University of Central Florida
27 PUBLICATIONS **0** CITATIONS

Multi-Threaded Sudoku Solver

Gangadhar Mahadevan, Rohit Durvasula, Vishnu Vidyan, Ebin Scaria
Department of Electrical Engineering and Computer Science
University of Central Florida
Email: gangadhar@knights.ucf.edu, drohit@knights.ucf.edu,
vishnu4v5@knights.ucf.edu,ebin@knights.ucf.edu

Abstract—**Sudoku is a popular logical numeric puzzle game where the user tries to solve by filling in numbers in an NxN matrix such that each row, column or sub matrix may contain only one instance of the numbers from 1 to N. Applications of solving such a puzzle can be found in areas of image encryption, digital watermarking, steganography, etc. In this paper we have implemented a serial as well as parallel algorithm which are capable of solving a Sudoku puzzle. The goal of this paper is to use the concepts we have learned in the Multicore programming course as part of our Masters program to efficiently parallelize a not so complex but challenging and a very interesting problem. For this reason we had taken up the Sudoku puzzle problem. [14] We have implemented both the serial as well as the parallel approach so that we can compare the average speed up the parallel approach gains over the serial counterpart. We ran experiments of various Sudoku puzzles of different sizes and complexities and our Parallel Sudoku Solver implementation was able to achieve a speed up of 3.5, based on puzzle solving/execution time, as compared to Serial Solver algorithm.**

I. Introduction

Sudoku is a number puzzle which in recent times has gained popularity. Sudoku has been coming up in various newspapers as a puzzle along with others such as crosswords and word search puzzles. The word ‘Sudoku’ literally means “the numbers must occur only once.” [15] This is what the rules of Sudoku are based on. A typical Sudoku puzzle consists of a grid of 9x9 numbers where the numbers belong to the set 1, . . . , 9. This grid is further divided into sub-grids of 3x3. These sub-grids along with the columns and rows are considered to be the regions of a Sudoku puzzle. A player trying to solve the Sudoku puzzle should make sure each of these regions contain the numbers 1-9 exactly once. This is the only rule that needs to be followed when solving a Sudoku puzzle. [1] However the entire Sudoku grid won’t be completely empty. Perhaps some of the blocks will be filled with digits which act as hints for the user to solve.

One important condition that is mandatory for a Sudoku puzzle is that it should have a unique solution. This is another reason why hints are provided. A Sudoku puzzle input is valid only if these hints are placed in such a way that there is always a unique solution corresponding to Sudoku puzzle. This is one of the critical factor under which we have designed the serial and parallel Sudoku puzzle solver. In this paper we have implemented a serial approach to solve the Sudoku puzzle. This algorithm is based on the approach suggested by Peter Norvig [24]. The serial algorithm provides us with a benchmark to analyze the performance build upon with the rest of our implementation [12], [16], [17], [21].Concurrency today is used in various areas of life because it provides significant performance improvements over serial applications [3], [6], [7], [26]. We have tried to parallelize this algorithm using pthreads [2] and message passing. The benchmarks obtained from the serial approach is then used to compare the efficiency of the parallel approach. The parallel approach utilizes concurrent threads to process the search space of the Sudoku thereby resulting in reduced processing time. When comparing the parallel implementation to the serial approach we were able to get an average speed up of 3.5x. This paper has also helped us in understanding the various concepts of threading and synchronization taught in the course.

The paper is organized as follows: Section 2 gives an overview of the various terminologies and rules to be considered when solving a Sudoku puzzle. Section 3 details the serial approach and implementation overview of solving the problem. Section 4 describes the approach we have followed for parallelizing the serial algorithm. In section 5 we compare the results of the serial approach with the parallel one to identify if we are able to achieve any significant speed up. Section 6 concludes the paper and outlines future work.

II. OVERVIEW OF SUDOKU

In this paper we have mostly used 9x9 grid Sudoku puzzles as the input for testing the solver implementation since these are the most widely used ones. Fig1 is an example of a 9x9 Sudoku puzzle before and after

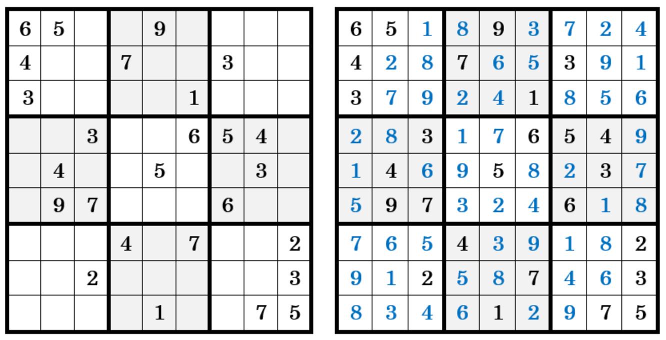

Figure 1: Solving Sudoku Puzzle

solving. The unsolved Sudoku puzzle with hints is taken as input. We process these hints provided to predict the rest of the missing numbers to solve the entire puzzle.

Some of the common terminologies we have followed here are described in Fig2. Each individual unit is termed as a cell. For every particular cell there will be 8 adjacent row cells, 8 adjacent column cells and 4 neighbouring cells in the sub-gird of 3x3. [28] These 20 cells (8 row cells + 8 column cells + 4 sub grid cells) are termed as the peers of the particular cell in consideration. When we start solving the puzzle, each of these cells will be checked against its peers to arrive at the cell's expected value. Each cell in a Sudoku puzzle can have up to 9 possible values. However a typical Sudoku puzzle usually will have some of the values already present. The particular cells will have a single possibility. Such cells are called singleton cells. The main objective of a solver would be to derive such singleton values from the already existing singletons present in the given Sudoku puzzle. For this purpose we refer to the set of rules mentioned by Peter Norvig's serial solver algorithm. [2] This is referred to as constraint propagation.

A. Step1: Constraint Propagation

The constraint propagation approach suggested by Peter Norvig comprises of two rules:

Rule 1: For any particular cell, if the number of possibilities can be reduced to a single value, then it can be converted to a singleton cell. This is illustrated in Fig. 3. Initially for the given 4x4 Sudoku puzzle we assign the possibilities (1, 2, 3, 4) for each of the non-singleton cells. For cell C4 the possibilities 2, 3 and 4 can be eliminated based on its peers value. This will leave cell C4 with the only possibility of 1. Using Rule1 C4 can then be assigned the value 1 thereby converting it to a singleton cell.

Rule 2: For any given cell after considering all its peers values if the possibility of a cell having a particular value is obvious or solitary, then the value can be assigned to the cell removing the rest of possibilities. This is illustrated in Fig.4. For cell A1, none of its peers has the possibility of having the value 4. Then A1 is identified as a singleton cell assigning it the value 4 irrespective of any other possibilities (value:1) it had previously.

Once a singleton cell is derived using rule 1 and rule 2, this can be used to further eliminate the possibilities of other peer cells. Likewise one could continue this process and eventually solve for all the non-singleton cells. However this is not the case always. For certain category of hard Sudoku puzzles, even after employing repeated constraint propagations we might not able to solve the puzzle. After a certain set of constraint propagation steps we might reach at a point where we are not able to further find any more singletons. Fig.5 shows such an example. The particular Sudoku puzzle after undergoing constraint propagation is not able to find any more singletons. When such a point is reached, a different approach is employed to find the next singleton. This is the search step.

B. Step2: Search

Once the constraint propagation step hits a dead end and is not able to find anymore singletons, the search step is carried out. Search is a form of intelligent guessing that is used to find the correct singleton values. In search one of the non-singleton cells is taken. We create multiple puzzle instances using all the possibilities of the particular cell. Using these various instances of the puzzle we can try solving the rest by employing the normal constraint propagation. Of all these possibilities, one

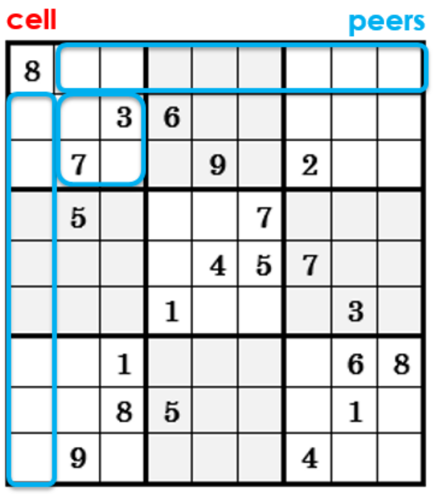

Figure 2: Sudoku Terminology

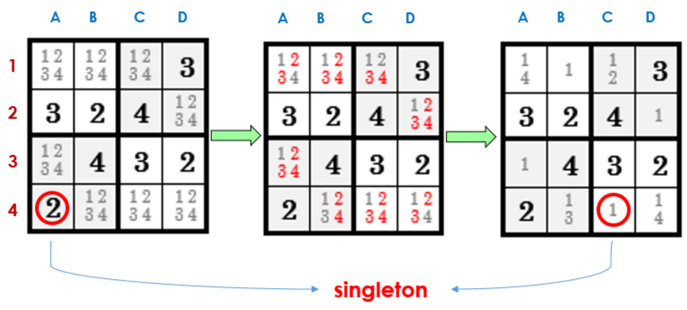

Figure 3: Using Constraint Propogation - Rule 1

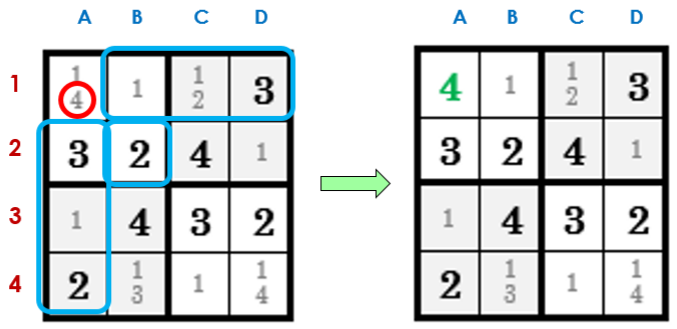

Figure 4: Using Constraint Propogation - Rule 2

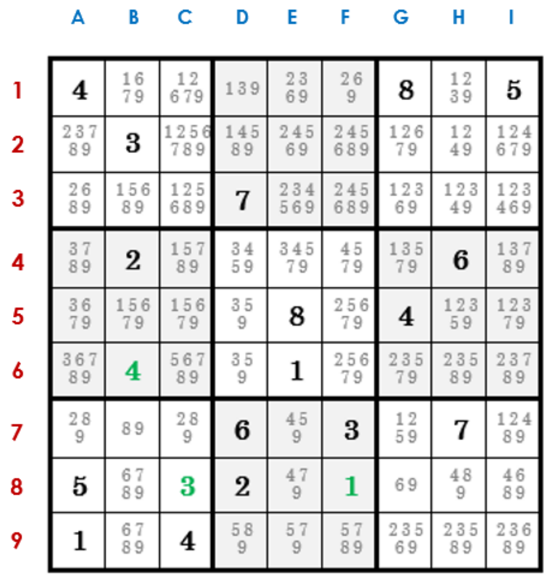

Figure 5: Unable to proceed after Constraint Propogation

would be correct. The single correct possibility will help us in arriving at the correct puzzle solution eventually. For example in Fig.5 cell B7 has two possibilities 8 and 9. Both the instances of the puzzle is taken; one with C7 as 8 and other with C7 as 9. Solving both these using constraint propagation one of these choices will eventually lead to a correct puzzle solution.

III. SERIAL IMPLEMENTAION

A. Serial approach

For the serial implementation of solving the Sudoku both constraint propagation and search are performed to eventually arrive at the solution. We start with the constraint propagation step and once a dead end is reached we employ the search approach. Using search we form multiple instances of the puzzle and based on this we further perform constraint propagation on these individual puzzles. If any of these assumptions were wrong, it will hit a contradictory situation which will violate the basic rules of the Sudoku puzzle. However a single search step would not always guarantee a solution of the Sudoku puzzle. A dead end might again be encountered when performing constraint propagation. At such a point search is again employed to move forward. If at any point any of these instances encounter a contradiction, the particular instance of the puzzle is ignored and the next possibility is considered. In this way the serial approach is depth first recursive search. Fig.6 shows an example of the Sudoku search tree. When selecting a particular cell for search we take up a cell which has least number of possibilities. This is due to the reason that when there are lesser possibilities a larger part of the search space will be eliminated and contradiction can be detected a lot earlier.

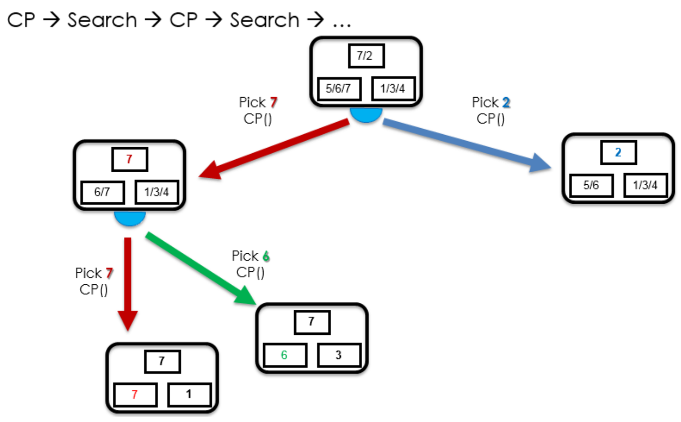

Figure 6: Serial Implementation

B. Problem representation

One of the major challenges we faced when implementing the constraint propagation and search approaches was coming up with an efficient way to represent the puzzle and the possibilities of each cells. Constraint propagation would have to store the list of possibilities of a particular cell. We could easily represent these possibilities using a simple data structure [5], [8], [13], [19], [25], [27]. In such a scenario we would need arrays to represent the possibilities of each of the 81 cells (for a normal 9x9 Sudoku). For the search step this would increase radically as more and more copies would be required. This would lead to the consumption of lots of memory and might limit us from solving the harder Sudoku problems which might have lot more possibilities in the search step. In this case we might have to be concerned about resource allocation [33].For this purpose we stored the possibilities as a binary representation of zeros and ones. Fig.7 illustrates this using the example of constraint propagation. Here cell D1 and D2 are represented in binary formats as 0100 and 1111 respectively. 0100 for D2 translates as possibility of having 4 as false, possibility of having 3 as true, possibility of having 2 as false and possibility having 1 as false. Here D2 is compared with D1 which is its singleton peer to eliminate the possibility of 3. For this the operation D1 AND D2 is performed which removes the possibility of 3 from D2. Hence the binary notation has helped in reducing the number of steps required for comparison and update the possible values of the cells significantly.

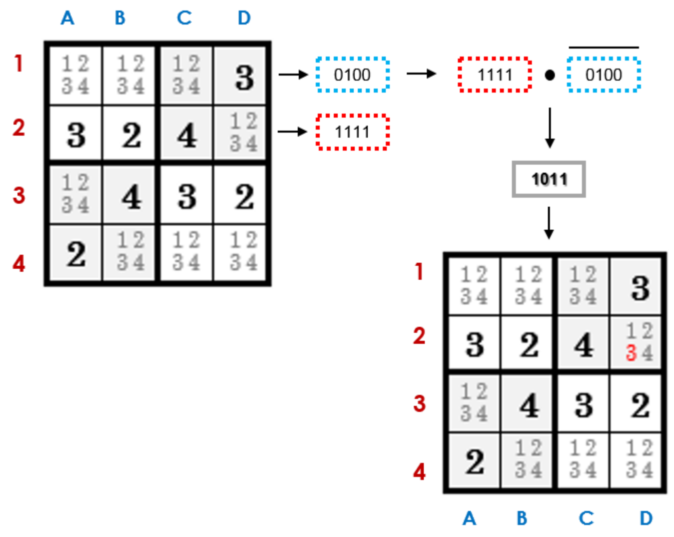

Figure 7: Cell Representation

IV. PARALLEL IMPLEMENTATION

A. Overview

We know that the solving time, or technically execution time, of solving a Sudoku problem depends on the complexity of the Sudoku itself. Studies show that simple Sudoku problems can be solved by using constraint propagation alone. But recursive constraint propagation and search together is needed for solving harder Sudoku puzzles. The number of recursions also vary for varying levels of puzzle difficulty. It is best to parallelize Sudoku problems because it will lead to a lesser solving time compared to sequential algorithm.Due to the significant advancement of concurrent data structures and concurrent techniques we have a lot of resources at our disposal to parallelize the problem [4], [9]–[11], [13], [18], [20], [20], [22], [29], [31], [32]. But spinning more threads without anticipating the consequences is not a good idea. We have parallelized the search part of the algorithm alone for couple of reasons. Constraint propagation is more strongly connected and has dependencies. So it is difficult to parallelize this part as threads would spend more time to communicate frequently rather than utilizing time for solving the problem itself. If a thread updates the list of possible values of a particular block in constraint propagation it has to pass on the information to all other threads before proceeding to next step. This makes it a poor candidate for parallelization. Search on the other hand is an independent task where each thread can solve in a sequential manner without worrying about other threads' execution status. If a thread finds the unique solution it signals all other threads that the problem is solved.

The advantage of parallelizing the search part is illustrated using Fig.6. In the serial approach the solver would have to scan through an entire branch of the DFS serach tree before proceeding to the other sibling branches of the tree to find the solution. So the final solution can be arrived at only be traversing thorugh all of the preceding sibling branches. By parallelizing serach of the tree, concurrent threads could work on different branches to achieve the solution quicker.

B. Message passing

As much as we are concerned with the communication overhead between threads which might increase overheads, we cannot completely ignore the case where threads need to have communication. [30] Each thread takes a possible value from the Constraint propagation and starts solving but only 1 thread will find the solution. Other threads might exit the process when they hit a dead end, i.e. when they realize the solution path taken is not the right one. There could be cases where the time taken to start and terminate the thread once dead end is detected might be more than the task performed by thread itself to solve the puzzle. So to make efficient use of threads we have communication of a thread to one of its adjacent neighbour in a round robin manner.

We set a Boolean flag to see if a thread is busy or not. By default it is set to false and whenever a thread hits a dead end it checks if the puzzle is solved. If not, it acknowledges to take some of the neighbouring

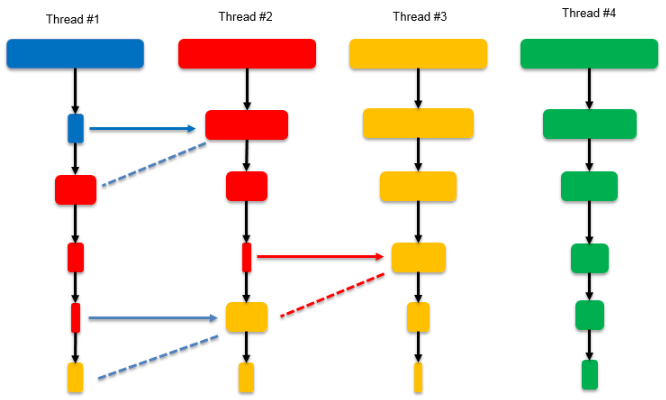

Figure 8: Message Passing

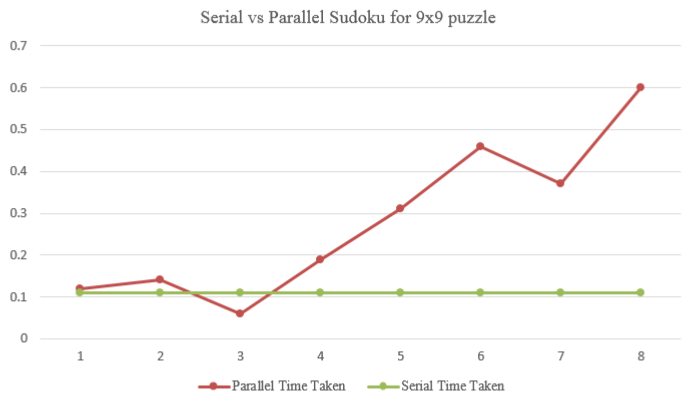

Figure 9: 9x9 Performance

threads work. The neighbouring thread splits its current work into two and assigns half of the work to the other thread. Following this approach, threads are prevented from exitting early and remaining idle. Now each thread will start solving the split puzzle individually. In Algorithm 1 a high level pseudocode of the message passing mechanism where adjacent threads communicate and continue solving the puzzle is mentioned. Since the puzzle is stored in a list represented as doubly linked list, the above code snippet is expected to take O(n) operation as each push and pop from head takes O(1) time.

V. Results

For testing the serial and parallel implementations of the Sudoku algorithm we have used a QuadCore, Intel i7 machine. We used a moderately hard 9x9 Sudoku puzzle with 3x3 mini grids during the implementation phase for testing out the code. In this itself, using just 4 threads we were able to observe significant change between the serial and parallel approaches. For carrying out a through performance analysis we ran the algorithm against puzzles of different difficulty. [23] employing different number of threads. The different Sudoku 9x9 puzzle difficulties are shown in table1.

Table I: Sudoku puzzle difficulty types

Difficulty Level	Number of Clues
Extremely Easy	More than 46
Easy	36-46
Medium	32-35
Difficult	28-31
Evil	17-27

Chart 1 shows the performance evaluation for the serial and parallel algorithms with varying number of threads. For this we have taken the average run time performance of Sudoku puzzles of varying difficulty. From the figure we can infer that as the number of threads increase from 1 through 3, the performance increases.

However when the threads further increase, the performance decreases. This could be attributed to the contention between threads that increase with increasing number of threads. For a 9x9 Sudoku puzzle we were able to attain most efficient run time performance when employing 3 threads.

We had also tested the performance for puzzle of size 16x16 which are referred to as the super Sudoku puzzles. Similar to the earlier 3x3 puzzles, for parallel implementation we were able to observe an average run time performance that was significantly better than the serial counterpart. Here too the performance increases as the thread count increase up to 3 threads. Beyond 3 threads the performance starts deteriorating owing to the issue of contention. Chart.2 shows the average run time comparison for a 16x16 puzzle.

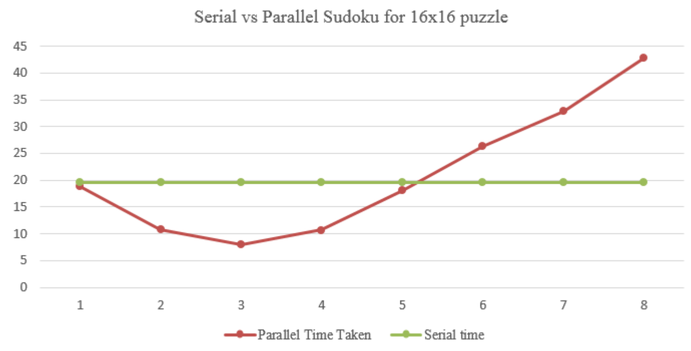

Figure 10: 16x16 Super Sudoku Performance

VI. Conclusion and Future Work

Based on our initial analysis and implementation techniques our work can be extended in various ways to improve the performance. One technique is to improve the

Algorithm 1 Message Passing

```
1: procedure ParallelSearch($puzzleToSolve$)
2:     $puzzleSolved \leftarrow false$                                    ▷ puzzle not solved
3:     $requestWork \leftarrow false$                                     ▷ Boolean flag
4:     while ( do$\neg puzzleSolved \wedge \neg threadAboutToExit$)       ▷ when more work is to be done
5:         $currentPuzzleSize \leftarrow globalPuzzleSize[threadID].size()$    ▷ find the current branch size
6:         if $currentPuzzleSize > 1$ then                    ▷ if branch size ¿ 1, still children are left
7:             $i \leftarrow 0$
8:             while $i < currentPuzzleSize/2$ do
9:                 $globalPuzzleList[(threadID - 1 mod totalThreads)].push(globalPuzzleList[threadID])$
10:                $global_puzzle_list[threadID].pop(global_puzzle_list[threadID])$
11:                $i \leftarrow i + 1$
12:            end while
13:            $requestWork \leftarrow false$                             ▷ reset Boolean flag
14:        end if
15:        $ConstraintPropogation()$                           ▷ carry on individual CP operation
16:        if $numSingletons == size * size$ then     ▷ if this CP has resulted in the solution work is done
17:            This CP has found the solution. Puzzle solved
18:            $puzzledSoved \leftarrow true$
19:        else
20:            $pickSearchCandidate()$       ▷ carry on Search Part individually by each thread to find solution
21:        end if
22:    end while
23: end procedure
```

current message passing by having interaction between multiple threads than just the neighbouring thread. In this way, the wait time of threads can be minimized and utilization can be improved. Another observation is for all our experiments when number of threads is equal to 3, the results seem to be the most optimum. As the number of threads were further increased the performance was hampered. These tests could have been carried out in more powerful systems as we had limited hardware resources.

Improvements could have been made in memory management. Currently we make duplicate copies of the puzzle for every search candidate. Instead we could maintain the puzzle in a global list and have multiple threads use the same shared puzzle. Our work can be further extended to solve multiple Sudoku puzzles at the same time so that we can further bring down the thread down time. We implemented message passing technique based on recursive constraint propagation and search methods. Simple Sudoku puzzles can be solved by using constraint propagation alone. Strongly connected dependencies amongst Sudoku cells makes it difficult to parallelize the constraint propagation. By parallelizing the Search part alone we were able to get a speedup of 3.5x on average. We hope our fundamental research can be further improved for solving larger puzzles and better results could be achieved in future works.

References

[1] Berggren, P., and Nilsson, D. A study of sudoku solving algorithms. *Royal Institute of Technology, Stockholm* (2012).

[2] BUTTLAR, D. A., Nichols, B., Buttlar, D., Farrell, J., and Farrell, J. *PThreads Programming: a POSIX Standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[3] Cook, V., Painter, Z., Peterson, C., and Dechev, D. Read-uncommitted transactions for smart contract performance. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems* (Dallas, TX, July 2019).

[4] Dechev, D., LaBorde, P., and Feldman, S. Lc/dc: Lockless containers and data concurrency: A novel nonblocking container library for multicore applications. *IEEE Access 1* (September 2013), 428–435.

[5] Dechev, D., Laborde, P., and Feldman, S. D. Lc/dc: Lockless containers and data concurrency a novel nonblocking container library for multicore applications. *IEEE Access 1* (2013), 625–645.

[6] Dechev, D., and Stroustrup, B. Reliable and efficient concurrent synchronization for embedded real-time software. In *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology* (2009), IEEE, pp. 323–330.

[7] Dechev, D., and Stroustrup, B. Scalable nonblocking concurrent objects for mission critical code. In *Proceedings of 24th International Conference on Object-Oriented Programming Languages, and Applications* (Orlando, Florida, October 2009).

[8] Feldman, S., LaBorde, P., and Dechev, D. Concurrent multi-level arrays: Wait-free extensible hash maps. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2013), IEEE, pp. 155–163.

[9] Feldman, S., LaBorde, P., and Dechev, D. Tervel: A unification of descriptor-based techniques for non-blocking programming. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)* (2015), IEEE, pp. 131–140.

[10] Feldman, S., LaBorde, P., Dechev, D., and level Arrays, C. M. Wait-free extensible hash maps. In *Proceedings of the 13th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (Samos, Greece, July 2013).

[11] Feldman, S., Valeraleon, C., and Dechev, D. An efficient wait-free vefctor. *IEEE Transactional on Parallel and Distributed Systems 27*, 3 (May 2016), 654–667.

[12] Feldman, S., Zhang, D., Dechev, D., and Brandt, J. Extending ldms to enable performance monitoring in multi-core applications. In *Proceedings of the Monitoring and Analysis for High Performance Computing Systems Plus Applications* (Chicago, IL, September 2015).

[13] Feldman, S. D., Bhat, A., LaBorde, P., Yi, Q., and Dechev, D. Effective use of non-blocking data structures in a deduplication application. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity* (2013), pp. 133–142.

[14] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming, revised first edition*. Morgan Kaufmann, 2012.

[15] infoplease. Meaning of sudoku. http://www.infoplease.com/askeds/meaning-sudoku.html. [Online; accessed 2015-10-15.].

[16] Izadpanah, R., Feldman, S., and Dechev, D. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)* (2016), IEEE, pp. 43–52.

[17] Izadpanah, R., Feldman, S., and Dechev, D. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In *Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing* (York, UK, May 2016).

[18] LaBorde, P., Feldman, S., and Dechev, D. A wait-free hash map. *International Journal of Parallel Programming 45* (2017), 421–448.

[19] LaBorde, P., Lebanoff, L., Peterson, C., Zhang, D., and Dechev, D. Wait-free dynamic transactions for linked data structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores* (2019), pp. 41–50.

[20] LaBorde, P., Lebanoff, L., Peterson, C., Zhang, D., and Dechev, D. Wait-free dynamic transactions for linked data structures. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores* (Washington DC, USA, February 2019).

[21] Lebanoff, L., Peterson, C., and Dechev, D. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *IFIP International Conference on Distributed Applications and Interoperable Systems* (2019), Springer, pp. 19–35.

[22] Lebanoff, L., Peterson, C., and Dechev, D. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *Proceedings of the 19th International Conference on Distributed Applications and Interoperable Systems* (Lyngby, Denmark, June 2019).

[23] Maji, A. K., and Pal, R. K. Sudoku solver using minigrid based backtracking. In *2014 IEEE International Advance Computing Conference (IACC)* (2014), IEEE, pp. 36–44.

[24] Norvig, P. Solving every sudoku puzzle. *Preprint* (2009).

[25] Painter, Z., Peterson, C., and Dechev, D. Lock-free transactional adjacency list. In *International Workshop on Languages and Compilers for Parallel Computing* (2017), Springer, pp. 203–219.

[26] Peterson, C., and Dechev, D. A transactional correctness tool for abstract data types. *ACM Transactions on Architecture and Code Optimization (TACO) 14*, 4 (2017), 1–24.

[27] Pirkelbauer, P., Dechev, D., and Stroustrup, B. Source code rejuvenation is not refactoring. In *International Conference on Current Trends in Theory and Practice of Computer Science* (2010), Springer, pp. 639–650.

[28] Schlingloff, B.-H. Teaching model checking via games and puzzles.

[29] Sher, G., Martin, K., and Dechev, D. Preliminary results for neuroevolutionary optimization phase order generation for static compilation. 33–40.

[30] Sottile, M., Dagit, J., Zhang, D., Hendry, G., and Dechev, D. Static analysis techniques for semi-automatic synthesis of message passing software skeletons. *ACM Transactions on Modeling and Computer Simulation 26*, 4 (September 2015).

[31] Zhang, D., LaBorde, P., Lebanoff, L., and Dechev, D. Lock-free transactional transformation. *ACM Transactions on Parallel Computing 5*, 1 (June 2018).

[32] Zhang, D., Laborde, P., Lebanoff, L., and Dechev, D. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC) 5*, 1 (2018), 1–37.

[33] Zhang, D., Lynch, B., and Dechev, D. Fast and scalable queue-based resource allocation lock on shared-memory multiprocessors. In *International Conference On Principles Of Distributed Systems* (2013), Springer, pp. 266–280.