# Parallelization of Sudoku

Alton Chiu (996194871), Ehsan Nasiri (995935065), Rafat Rashid (996096111)
{alton.chiu, ehsan.nasiri, rafat.rashid}@utoronto.ca
University of Toronto

December 20, 2012

## Abstract

*Sudoku is a logic-based number-placement puzzle game where the players goal is to complete a nxn table such that each row, column and box contains every number in the set $\{1, \ldots, n\}$ exactly once. In this report, we attempted to generate a highly parallelized Sudoku solver using the pthreads library on the Linux kernel. Based on our runtime evaluation, we achieved an average speedup of 4.6 times over the serial implementation by using locking and yielding methods with four threads. We believe further speedup can be gained by making better use of threads in our message passing version or by taking advantage of GPUs.*

## 1 Introduction

In this report, we present our implementation of a highly parallelized Sudoku Solver using the *pthreads* library on the Linux kernel. We begin with a discussion of the motivation for our work and present our contributions. Section 3 presents a detailed account of the work completed. In Section 4, we discuss our evaluation methodology and Section 5 presents our results. Section 6 presents related work in existing Sudoku and other logic game solvers and Section 7 provides a discussion of the future direction this project could take. Finally Section 8 concludes the report.

### 1.1 Motivation

This project is motivated by the increase in popularity of Sudoku and similar logic puzzles over the past decade [1]. Numerous serial Sudoku solver implementations exist and are readily available [2][3][4]. However, due to its recency, there is relatively little work on parallel implementations. Parallelizing a serial Sudoku solver can improve its speed and increase the viability of solving larger size Sudokus. The Sudoku solver algorithm also conveys some of the basic tradeoffs of parallel software development such as dependencies and work-sharing which are interesting to inspect and study.

### 1.2 Our Contributions

Our first contribution is the implementation of a Sudoku solver that explores the search space of the puzzle in parallel. This implementation allows each thread to work on a smaller search space in parallel and minimizes the execution time, leading to a solution faster. We achieved an average of up to 4.6 times speedup using 4 threads with our best parallel algorithm which uses fine-grain locking and yielding.

Our second contribution is the implementation and analysis of two major synchronization methods to parallel Sudoku solvers: locks and message passing. We provide runtime measurements of these algorithms run with up to eight threads.

Our third contribution is implementation and analysis of different methods of waiting for idle threads. We implemented three variants: spin-looping, yielding, and condition variable signalling. We provide the runtime measurements of these variants run with up to eight threads.

## 2 Overview of Sudoku

Sudoku is a logic-based number-placement puzzle game. The standard Sudoku puzzle is a table made up of 9 rows, 9 columns and 9, 3x3 boxes, as shown in Figure 1. The puzzle starts with given numbers in various positions and the player's goal is to complete the table such that each row, column and box contains every number from 1 to 9 exactly once.

Although the 9x9 variation of Sudoku is most common, larger variations exist to increase the puzzles difficulty. An example is a 16x16 variation made up of 16 rows, 16 columns and 16 4x4 boxes. Cells with a single possibility is called a singleton. By using specific rules and information from each cells peers, we can iteratively find more singletons to help complete the table. A cell's peers is defined by all other cells that belong to the same row, column or box. We describe the two steps we employ to solve a Sudoku puzzle below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 5 | | | 9 | | | | |
| 4 | | | 7 | | | 3 | | |
| 3 | | | | | 1 | | | |
| | | 3 | | | 6 | 5 | 4 | |
| | 4 | | | 5 | | | 3 | |
| | 9 | 7 | | | | 6 | | |
| | | | 4 | | | | | 2 |
| | | 2 | | | 7 | | | 3 |
| | | | | 1 | | | 7 | 5 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 5 | 1 | 8 | 9 | 3 | 7 | 2 | 4 |
| 4 | 2 | 8 | 7 | 6 | 5 | 3 | 9 | 1 |
| 3 | 7 | 9 | 2 | 4 | 1 | 8 | 5 | 6 |
| 2 | 8 | 3 | 1 | 7 | 6 | 5 | 4 | 9 |
| 1 | 4 | 6 | 9 | 5 | 8 | 2 | 3 | 7 |
| 5 | 9 | 7 | 3 | 2 | 4 | 6 | 1 | 8 |
| 7 | 6 | 5 | 4 | 3 | 9 | 1 | 8 | 2 |
| 9 | 1 | 2 | 5 | 8 | 7 | 4 | 6 | 3 |
| 8 | 3 | 4 | 6 | 1 | 2 | 9 | 7 | 5 |

Figure 1: 9x9 Sudoku Puzzle with solution

## 2.1 Constraint Propagation

Constraint Propagation (CP) is the first step used in solving a Sudoku. It consists of two rules:

**Rule 1:** For any cell, if a number already exists in its row, column or box (the cell's peers), the possibility of that number for that cell is removed.

**Rule 2:** For any cell, if all of its peers has a specific number removed, the cell itself must contain that number.

Figure 2 shows an example of Rule 1. The upper left cell has two possibilities: $\{8, 9\}$. Once the 9 is placed in one of the cells peers, it is removed from the possibilities of the upper left cell. This reduces its possibilities to only $\{8\}$, making it a singleton.

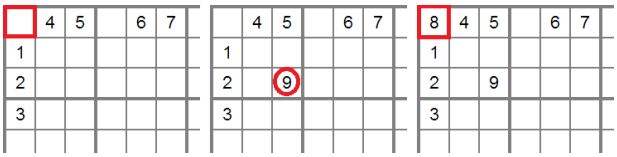


Figure 2: Example of Rule 1

Figure 3 shows an example of Rule 2. For the left Sudoku's top-left cell, all its peers (marked by blue) do not have 3 as a possibility. This means 3 can only be placed in the top-left cell, as shown in the puzzle on the right.

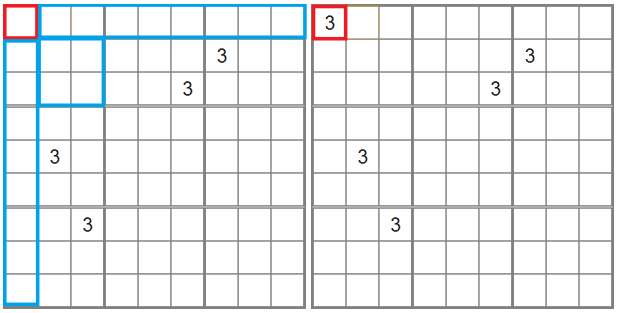


Figure 3: Example of Rule 2 - (blue cells are peers)

By repeatedly applying these two rules, the possibilities are gradually minimized and more singletons can be discovered. Once CP cannot reduce the amount of possibilities any further, the Search method is used.

## 2.2 Search

Here a non-singleton cell is chosen to assume one of its possible values. We then continue to perform Constraint Propagation with this assumption. If the assumption is correct, we will eventually arrive at the solution to the puzzle. However if a contradiction is reached, the chosen value is eliminated from the cell's possibilities and another possible value would be attempted for that cell.

# 3 Implementation Details

Figure 4 provides a high level overview of our work. The arrows illustrate the dependencies between the different components. The Parser is used to read and write Sudoku puzzles from a file. As an example, a 16x16 puzzle would contain numbers from 1 to 16, with 0s denoting the unsolved cells.

The Validator is used to verify both solved and unsolved Sudokus produced by our solvers and our Sudoku Generator. The Generator produces puzzles with unique solutions. Its implementation is described in Section 3.1. The Peers Generator produces the peers list given the size of the puzzle as its input, as the peers list is different for puzzles with different sizes. By generating the list of peers for each cell, it removes the need to calculate the indexes of each peer during the execution of the algorithm.

Finally, we have implemented a serial Sudoku solver and several parallel Sudoku solvers. Their design is described in Sections 3.2 and 3.3 respectively. All of our code is implemented in C++. Our parallel algorithms uses the *pthreads* library.

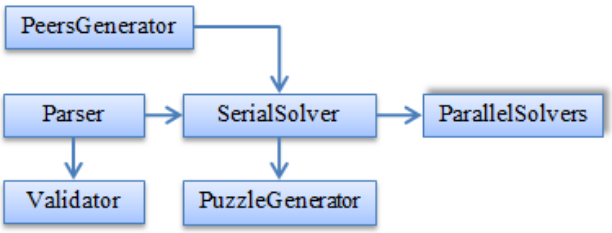


Figure 4: Implementation breakdown

## 3.1 Sudoku Generator

We created a Generator capable of creating new Sudoku puzzles automatically. The Generator requires a correctly solved puzzle as input where all the cells are already filled. As the first step, the Generator performs transformations

on the solved puzzle to create another solved puzzle. We utilize four such transformations described in [5] to swap around the puzzle cells. These are documented below:

1. **Swap cells (rows) horizontally:** rotating the grid around the middle row. The first row swaps with the last, the second swaps with the second last and so on. No change to the middle row in odd-sized puzzles.
2. **Swap cells (columns) vertically:** rotating the grid around the middle column. The first column swaps with the last, the second swaps with the second last and so on. No change to the middle column in odd-sized puzzles.
3. **Swap cells around the main diagonal:** rotating the grid around the main diagonal.
4. **Swap cells around the minor diagonal:** rotating the grid around the minor diagonal.

The second step is to remove as many of the filled-in cells as possible. The goal is to generate new unsolved puzzles with only a single solution. First, we randomly select a cell in the puzzle to remove. We know that the puzzle can be solved with that cell's value. Then we attempt to solve the puzzle with all other possible values. For a 9x9 puzzle, if we remove a "1" from the cell, we attempt to solve the puzzle with each value from 2 to 9. If it is solvable with another value, this means there will be multiple solutions if the value in the cell is removed. Otherwise, it can be safely removed. This process of removing cells is repeated until no further cell can be removed. At such point, the final unsolved Sudoku is generated.

These two steps can be repeated to create multiple Sudoku puzzles. Our implementation allows Sudokus of arbitrary sizes to be generated. We have validated our Generator with 9x9, 16x16 and 25x25 puzzles using our Validator. Our Generator provides us with three important guarantees that are useful when evaluating our solvers:

1. Generated puzzles have unique solutions.
2. Generated puzzles are 'evil', since we continue removing cells until no further cell can be removed.
3. Generated puzzles have the same time complexity as the solved input Sudoku due to the properties of the transformations.

## 3.2 Serial Solver

We start with a serial implementation of Sudoku in C++ based on Peter Norvigs algorithm [3]. The algorithm uses Constraint Propagation and Search discussed in Section 2. We describe their implementation in our algorithm below. After our algorithm finds the solution to the puzzle, we ensure it is correct using our Validator.

### 3.2.1 Constraint Propagation

We cycle through each cell, performing checks on the two rules presented in Section 2.1. For the first rule, for each peer, if it is a singleton, we remove the peers value from the possibilities list of the cell. When all the peers of a cell have been checked, we check the cell itself to see if it has become a singleton.

For the second rule, if a number is removed from all of its peers, we can safely turn the cell into a singleton with the value of that number. Any cell that becomes a singleton triggers a new round of CP. This means the cell will be removed from its peers possibility list – which may in turn make one or more of its peers singletons.

### 3.2.2 Search

Once Constraint Propagation cannot reduce the possibilities any further, we use Search. Here we try a possible value for a cell and see if that value causes the puzzle to reach a contradiction. So if a cell has three possible values, we create three copies of the current puzzle and force one of the possibilities for each child puzzle. One of the children will lead to the correct solution and the other two will reach contradictions. To find out, we perform CP and possibly Search on each child which can have their own children, resulting in a tree of candidate puzzles. One of leaves of the tree will contain the solution to the puzzle. The execution of the algorithm is illustrated below:

$$\text{CP}() \rightarrow \text{Search}() \rightarrow \text{CP}() \rightarrow \text{Search}() \rightarrow \ldots$$

When the algorithm wants to start performing Search, it can choose from any of the cells with multiple possibilities. Choosing this search candidate cell can have an important effect on performance. The best possible search candidate would be the cell with the fewest possibilities. This is because if a selected cell had four possibilities, by realizing that one is impossible, we only remove one-fourth of the search space (left tree in Figure 5).

Alternatively, if a selected cell had two possibilities, by eliminating one, we remove one-half of the search space (tree on the right in Figure 5) and guarantee us a correct singleton! Thus, we select a non-singleton search candidate with the fewest possibilities. One special case is if all possibilities list have one or fewer possibilities. This means the puzzle has reached a contradiction and it is removed from the tree.

## 3.3 Parallel Solvers

We have parallelized the Search portion of the serial algorithm for the following reasons:

Firstly, for simpler puzzles that can be solved using only CP, the runtime is very fast already.

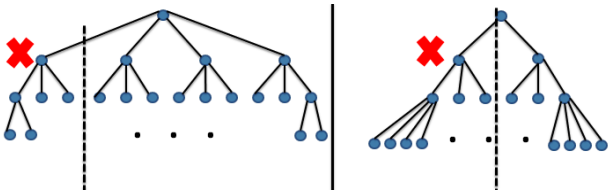

Figure 5: Selecting a good search candidate. Left: a contradiction leads to pruning a fourth of the tree. Right: a contradiction leads to pruning a half of the tree.

Secondly, strongly connected dependencies make it extremely difficult to parallelize Constraint Propagation. If a thread updates the possibility list of a cell during a step in CP, that information has to be propagated to all other threads before they can move forward to the next step in CP. This causes the threads to execute in lockstep instead of performing the computations in parallel.

Finally, search produces candidate puzzles that the serial algorithm attempts to solve (or discard) one by one. Since these puzzles can be solved independently, it makes sense to have multiple threads attempt to solve different candidates. When one of the threads finds the solution, it can tell the other threads to stop.

By parallelizing Search, each thread tries different branches in the search space. The immediate benefit is shown in Figure 6. In the serial implementation (left), the solver will have to search the entire left branch before finding the solution in the right half of the search space. By having a parallel implementation (right), the solution can be immediately found by a second thread.

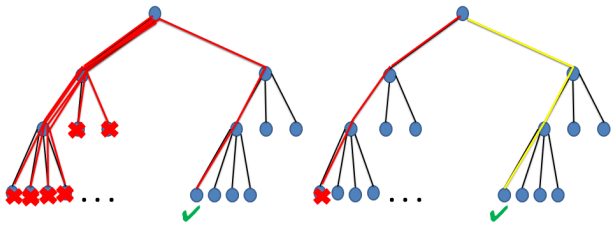

Figure 6: Benefit of parallelizing Search

However, the flipside may also be possible. If the solution is located in the first branch, the parallel algorithm may be slower due to the thread management overhead and executing multiple branches in parallel. However, the runtimes of these cases are very low in the first place, so the overhead becomes a small penalty when amortized over many puzzles.

Below, we describe the several variations of our parallel Sudoku solver algorithm.

### 3.3.1 Message Passing

For this version, each thread is initialized with an initial candidate puzzle inserted into its private puzzle list. Each thread then independently executes the serial algorithm on its candidate puzzle, with any child candidates resulting from performing Search being appended to the end of its own list.

This means that each thread will own a branch of the tree and all of its children. Each thread walks down its branch of the tree and explores all of its leafs in a depth-first fashion. As soon as a thread hits a leaf that contains the solution, it notifies all the others to stop.

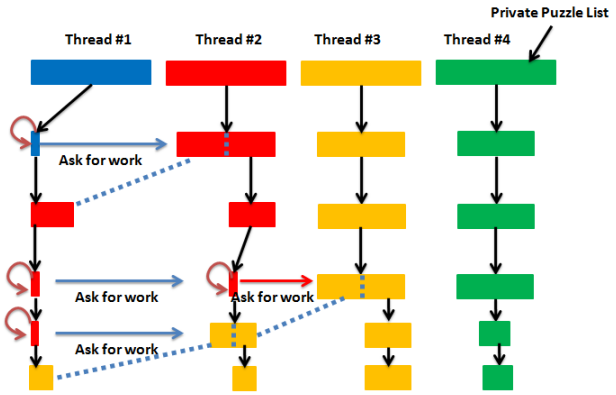


Figure 7: Our *message passing* implementation

Based on the algorithm described thus far, if one or more of the threads exhausts their private list before the solution is found, they would exit. Due to the disproportionate nature of the search space, this is not a very efficient use of the threads.

Instead of exiting, it can help another thread by taking over a subtree of another branch. A thread that runs out of work will therefore ask another thread for additional work by raising a boolean flag. Each such flag is shared between two adjacent threads (corresponding to their thread IDs). For example, with n number of threads, Thread 0 asks Thread 1 for work. Thread 1 asks Thread 2 and Thread n asks Thread 0.

When a thread sees that another thread has asked for work, it would pass half of its puzzle list to the other thread's list and then lower the flag. Since the list is implemented as a doubly linked list by the STL C++ library, the entire operation takes O(n) time as each insert and deletion from the head of a list takes O(1) time. This version of the algorithm uses regular boolean variables to perform the handshake needed to transfer work from one thread to another and busy waits on a while loop when waiting to be given work.

We have also tried using *thread_yield()* and condition variables to perform the transfer of puzzles between

thread lists. These are described next.

#### 3.3.2 Yielding

This is a variant of the Message Passing algorithm with a call to *thread_yield()* inside the wait loop instead of spinning on the flag variable. A thread calling *thread_yield()* ideally would relinquish the CPU and be placed in the back of the same static priority queue.

Ideally, a longer running thread would yield when it runs out of work because longer running threads in the Linux kernel have lower priority, yielding for shorter running threads with higher priority. Theoretically, this should work because longer running threads would go through more puzzles and be more likely to run out of puzzles than shorter running threads. However, this may not always be the case. A shorter running thread in the highest active priority by itself may still run out of puzzles. In this case, there would be no benefits to yielding, as it would yield and the scheduler would immediately reschedule the yielding thread.

#### 3.3.3 Signalling

This is a variant of the Message Passing algorithm in which instead of spinning on a variable, threads would go to sleep after asking another thread for work. When another thread finishes giving work to the sleeping thread, it signals the thread to wake up. This is implemented with *pthread_cond_t*.

#### 3.3.4 Using Locks

In this algorithm, we have used a single global puzzle list. This means the entire tree would be in the shared memory for each thread to work on. Since the puzzle list is shared among all threads, accesses are synchronized via a single *pthread_lock_t*. After acquiring a lock on the global puzzle list, each thread can:

- Take on a branch of the tree (pop an item out of list)
- Push back any child puzzles back into the global puzzle list (extend the tree)
- Take on other branches as soon as it reaches a contradiction on its own branch

The introduction of this global list will allow threads to work without interruption, without the need to ask for work from another thread and without waiting for other threads to respond. As long as the puzzle is not fully solved, any thread can take on a new branch of the tree. Two variants of this algorithm are fine-grained locking and coarse-grained locking.

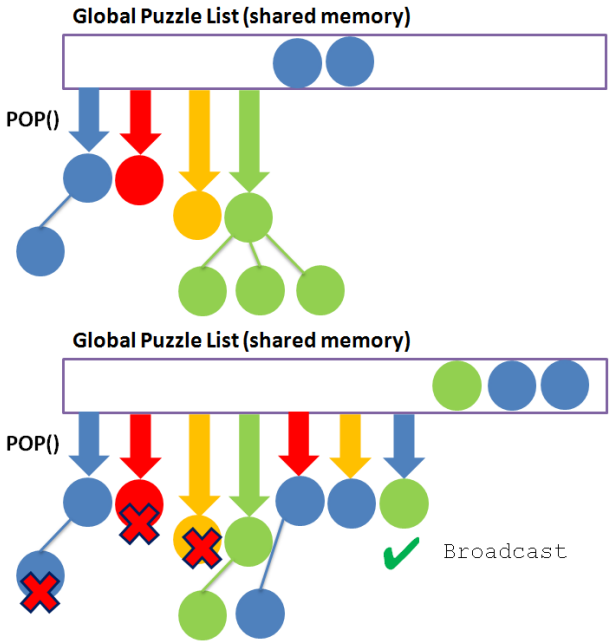


Figure 8: Our locking implementation: no thread will be idle at any time

For fine-grain locking, we acquire the lock before pushing or popping a puzzle from the list and release it right after. For the coarse grain locking, the critical section instead becomes the entire loop that performs the Search algorithm and pushes the children to the list.

## 4 Evaluation Methodology

We performed our evaluation on the Intel Core 2 Quad q9450 @ 2.66 GHz machines with 4 GB of RAM running the Debian Linux kernel 2.6.32. Due to the nondeterministic nature of multithreaded executions, runtimes can vary even when solving the same puzzle. We therefore averaged the results of a 100 16x16 evil puzzles, running them 10 times for each variant of our algorithms. We measure elapsed time using the *gettimeofday()* method which gives us accuracy to the microseconds. Our measurements are automated, requiring no human input. A Python script was used to run each variant and output the timing results to files.

Evil puzzles are created using our Sudoku Generator described in Section 3.1. These puzzles are dubbed evil because the Generator removes as many cells as possible with the guarantee that the solution to the puzzle is unique.

The 16x16 size puzzles proved to be the optimal size of Sudoku for measurement and analysis. The 9x9 Sudokus had very fast runtimes in the order of milliseconds. For these puzzles, the runtime variance was high, creating less meaningful results. The 25x25 evil puzzles re-

quired a large amount of puzzles to be added to the puzzle lists as Search dominated early on during the execution of the algorithm when the number of singletons was low. With these puzzles, we frequently observed failed *malloc()* calls due to the process running out of memory.

# 5 Evaluation Results

In this section, we compare the runtime results of our serial and parallel Sudoku solvers.

## 5.1 Runtime Performance

Figure 9 compares the average time it takes to solve a single 16x16 evil puzzle for our message passing (with spin looping) and locking algorithms with respect to the number of threads used. The runtime of our serial algorithm (17.5s) is on the graph for reference.

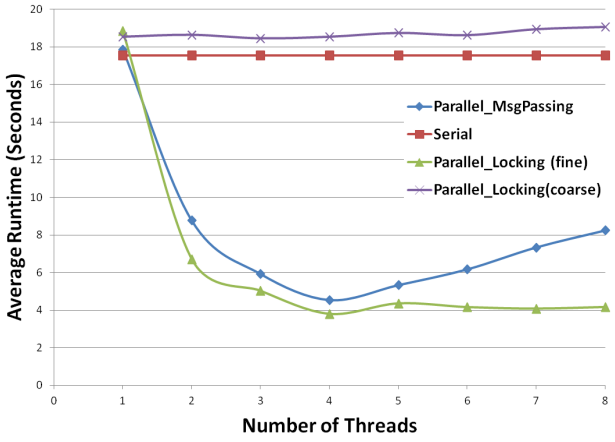


Figure 9: Average execution time for solving a single 16x16 evil puzzle, amortized across 100 different puzzles

An immediate speedup is observed for fine-grain locking and message passing algorithms as we add the second thread to perform work. For two threads, the message passing and fine-grain locking runtimes were an average of 8.9s (1.97x speedup) and 6.7s (2.61x speedup) respectively. This speedup is a direct result of the earlier access to other root branches of the decision tree where the solution may be.

Regardless of the number of threads used, our coarse grain locking algorithm performs worse than our serial implementation. The coarse grain locking took an average of 18.5s for two threads (a 5.7% slowdown) up to an average of 19s for eight threads (an 8.6% slowdown).

Since our coarse grain locking implementation locks the entire loop body, the Search algorithm becomes essentially serialized with one thread at a time accessing the puzzle list and performing work. By minimizing the area the lock covers, we can increase the code that can be accessed by multiple threads in parallel and increase the speedup. This shows that by choosing a locking method that is easier on the programming, the resulting runtime can be worse than the serial implementation. Creating more threads results in the same serial-like execution with more overhead from thread creation and locking calls, resulting in a larger runtime.

For message passing, after four threads, the runtime increased. To avoid possible race cases, a thread only passes puzzles to subsequent threads. This implementation can cause cascading effects. Consider if both Thread 2 and 3 run out of puzzles. Thread 3 requests work from Thread 2, who has no work to give. Thread 2 requests work from Thread 1. Thread 1 gives half of its work to Thread 2, who subsequently gives half of its work to Thread 3.

This increases the time it takes for Thread 3 to receive work, and also decreases the amount of work that Thread 3 receives. This also increases the idling time of threads as they wait to be given work. This could be problematic with a larger number of threads where it is possible that a large amount of contiguous threads can run out of work. This results in an increase in the runtime for a larger number of threads for message passing.

For finer-grain locking, the algorithm strives to keep all threads busy and have no idle threads and therefore there was a great benefit as the number of threads was increased up to four threads. Beyond four, the benefits stopped as the locking becomes a bottleneck. The figure ultimately shows that the fine graph locking performed best, as the number of threads idling was minimized.

## 5.2 Yielding Performance

Figure 10 compares two of our message passing algorithms: yielding and busy-wait looping. For lower number of threads, yielding provides no benefit. This is because with a low thread count, there is a greater chance that the thread that yields is the one with the highest priority. Hence, it will be rescheduled after a yield. We observe a benefit for a larger number of threads because there is a greater chance that the thread who yields is not at a high priority and that there is another thread in the same priority to take its place if needed.

## 5.3 Condition Variable Performance

Figure 11 shows our condition variable runtime compared to our yielding implementation. Our condition variable implementation actually performs worse than our yielding implementation. This is because using condition variable signalling is very expensive [6]. This proved to be very costly for our Sudoku solver. However, some applications cannot avoid the use of signalling. Our application

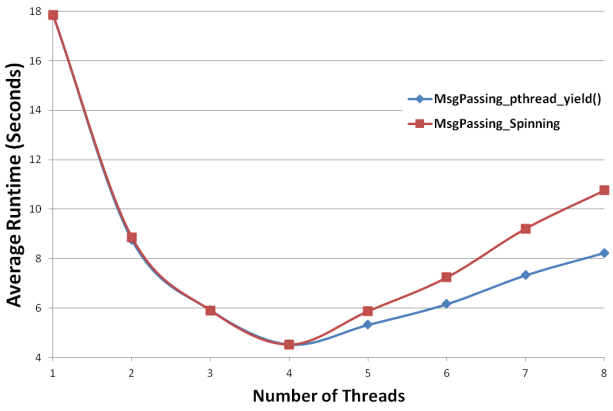


Figure 10: Yielding Results

was simple enough to get away with using a few simple boolean variables.

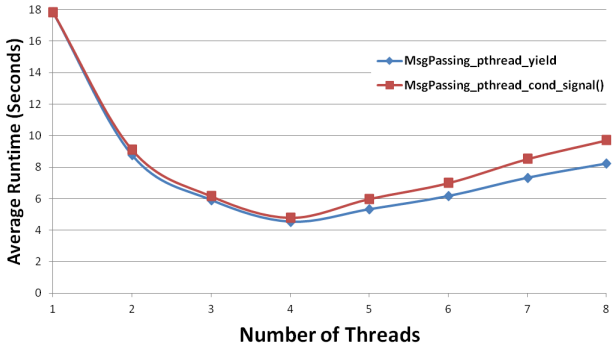


Figure 11: Condition Signaling Results

# 6 Related Work

## 6.1 Existing Sudoku Solvers

There are existing methods that solve a Sudoku using different strategies from the one we employ. Dan Taylor employs a DFS strategy similar to our Search phase by trying different possible values [2]. However, his solution does not employ Constraint Propagation and uses recursion to backtrack when his algorithm reaches a contradiction.

Bill DuPree employs an iterative strategy starting with Constraint Propagation like our solution [4]. With no more changes, two more advanced rules called naked/hidden subset and chute alignment are employed to eliminate more possibilities. After this, a recursive, trial and error with backtracking is used to randomly try remaining possible values like Dan Taylor. Differently from our solution, Bill DuPree does not go back to Constraint Propagation after assuming a possible value for a cell. This means the search space can become exponentially larger if the previous steps does not eliminate most of the possibilities.

In theory, we could have alternated between Constraint Propagation with the two advanced rules and Search. However, the advanced rules are also difficult to parallelize and would make the serial component of our parallel algorithm larger. Exploring this could be the subject of future work.

Finally, Russ Klein implemented a parallel solver using *pthreads* [7]. His implementation created a thread representing each possible candidate value and a copy of the same puzzle for each thread. Each thread only looks through the puzzle to contribute with its possible value. At the end of each iteration, a master thread combines all the new information from each thread and redistributes the new partially-filled puzzle before the next iteration. This implementation requires each thread to work on the same puzzle and coordinate between iterations whereas our implementation allows each thread to work on a slightly different puzzle on its own.

## 6.2 Work on Parallelizing Logic Games

Sudoku has given rise to other logic games and brain-teasers in the past decade [1]. Like solving Sudoku, the logic and strategy in each of these games can be translated into rules and converted into automated solving programs. A popular puzzle in recent years is Kakuro, a number-logic puzzle where each row and column in a grid must sum up to an indicated value. Kakuro can be reduced to a constraint problem [8] or be solved in a manner similar to solving crossword puzzles [9]. Work has been proposed to compare a sequential Kakuro solver with a parallel implementation [10].

Other solvers for puzzles such as Str8ts, a similar logic-number puzzle where each number in a row or column must be in sequential order [11] and Hidato [12] also exist. Solving these are NP-complete and take an exponential time to solve with respect to the puzzle size. The relative recency of these puzzles has resulted in little work in parallelizing these solvers and is an interesting challenge worth exploring.

Beyond number logic puzzles, there has been work on parallelizing other logic games and applications. Conways game of life is an application with rules that allow for the growth of a species of pixels [13]. by letting the program run, the visual output can fade away, become stable, or oscillate between patterns. Conways game of life has been a standard example of the speedup of parallelization. Since many of the rules can be run in parallel, large scale implementations can even be passed to the GPU for more massive parallelization.

Work in parallelizing Chess work began as early as 1985 [14] where chess was attempted to be solved by

using a search space to determine the long term consequences of each move in parallel and choosing the move with the lowest risk. StarTech expands on parallel chess algorithms by running the program on 512 processors [15], successfully finishing third at the 1993 ACM International Computer Chess Championship. The most famous chess playing computer may be Deep Blue, developed by IBM [16]. Deep Blue successfully defeated World Chess Champion Garry Kasparov in 1997 with its massively parallel code system containing multiple levels of parallelism.

# 7 Future Work

Here we provide a few possible directions in which the work presented in this report can be extended. One task we could attempt is to possibly improve our message passing algorithm so that any working thread can transfer work to any thread asking for work to prevent the cascading effects observed in our current implementation. We believe this would work better in minimizing the time a thread waits for work, with the trade off of a possible bottleneck when multiple threads attempt to give work and fight for a lock.

Another possibility would be to see the impact of using a larger number of threads. For this to be beneficial, we may need to test on a machine that can take better advantage of a larger number of threads. We were limited to quad core machines in our evaluation. A higher number of cores would allow for more threads and possibly better parallel execution.

Problems encountered with memory allocation for larger sized puzzles could possibly be fixed with a different implementation that would be more conservative with puzzle creation. For example, instead of duplicating the puzzle for each search candidate, we could save only the difference between two puzzles. Using this method, upon a contradiction on one branch of the tree, we can traverse back up the tree and re-generate the puzzle using the diffs saved.

We limited our implementation to solve puzzles sequentially, with subsequent puzzles being solved after taking measurements. We can improve the performance by attempting to solve multiple puzzles in parallel at the same time. This would reduce thread downtime, as unused threads could begin attempting to solve subsequent puzzles, amortizing the overall runtime.

Beyond our implementations, we did not attempt to use GPUs, OpenCL and FPGAs to accelerate our solvers. We could attempt to try solving many puzzles at the same time with GPUs in the future. We believe this may be very difficult, as the overhead of GPUs is only paid for when we are working on hundreds of threads at the same time and GPUs excel when many of the tasks are similar in nature. From our experience, it may be difficult to predict when Sudoku solvers would need Constraint Propagation or Search methods, and Sudoku solver runtime can vary depending on the puzzle.

# 8 Conclusion

This report introduces several parallel implementations of a Sudoku solver based on Constraint Propagation and Search methods. Strongly connected dependencies made it extremely difficult to parallelize CP. Traversing the solution space tree during a Search in parallel is the best way to reach a solution faster. We implemented three message passing methods with the major difference in how a thread handles waiting for work: spin-looping, yielding, or waiting on a condition variable. We also implemented two lock-based methods: one with a coarse-grain lock over the entire loop body, and one with a finer-grain lock around the accesses to a global puzzle list.

For most puzzles, we observed an immediate speedup with most methods by increasing the number of threads to two. We achieved an average of up to 4.6 times speedup with our best parallel algorithm which uses fine-grain locking with yielding and four threads. Using coarse-grain locking, expensive condition variable calls, or spin looping algorithms increased the execution time. Overheads caused by increasing the number of threads beyond four did not allow the runtime to decrease any further.

# References


[1] M. Fackler, "Inside japans puzzle palace." [Online]. Available: http://www.nytimes.com/2007/03/21/business/worldbusiness/21sudoku.html?pagewanted=all&_r=0

[2] D. Taylor, "Solving every sudoku puzzle," *Logical Genetics*. [Online]. Available: http://logicalgenetics.com/showarticle.php?topic_id=1624

[3] P. Norvig, "Solving every sudoku puzzle." [Online]. Available: http://www.norvig.com/sudoku.html

[4] B. DuPree, "A sudoku solver in c." [Online]. Available: http://www.techfinesse.com/game/sudoku_solver.php

[5] DryIcons, "A simple algorithm for generating sudoku puzzles." [Online]. Available: http://dryicons.com/blog/2009/08/14/a-simple-algorithm-for-generating-sudoku-puzzles/

[6] C. Honess, "Techniques for improving the scalability of applications using posix thread condition

variables," p. 5, February 2007. [Online]. Available: http://h21007.www2.hp.com/portal/download/files/unprot/hpux/MakingConditionVariablesPerform.pdf

[7] R. Klein, "Dude, wheres my multi-core performance?" [Online]. Available: http://blogs.mentor.com/russ_klein/blog/2011/01/10/dude-wheres-my-multi-core-performance/

[8] H. Simonis, "Kakuro as a constraint problem," *Proc. seventh Int. Works. on Constraint Modelling and Reformulation*, 2008.

[9] K. Hiddemann, "A simple kakuro solver." [Online]. Available: http://meilof.home.fmf.nl/2008/11/30/a-simple-kakuro-solver/

[10] T. G. Eamon Doran, Michael Wezalis, "Comparison of sequential and parallel kakuro solver to highlight speedup." [Online]. Available: http://www.cs.rit.edu/~ark/531/team/u9/proposal.pdf

[11] A. Stuart, "Str8ts solver." [Online]. Available: http://www.str8ts.com/str8ts.htm

[12] R. C. editors, "Solve a hidato puzzle." [Online]. Available: http://rosettacode.org/wiki/Solve_a_Hidato_puzzle

[13] J. Conway, "The game of life," *Scientific American*, vol. 223, no. 4, p. 4, 1970.

[14] M. Newborn, "A parallel search chess program," in *Proceedings of the 1985 ACM annual conference on The range of computing: mid-80's perspective: mid-80's perspective*. ACM, 1985, pp. 272–277.

[15] B. Kuszmaul, "The startech massively parallel chess program," *Journal of the International Computer Chess Association*, vol. 18, no. 1, pp. 3–19, 1995.

[16] M. Campbell, A. Hoane, and F. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1, pp. 57–83, 2002.