Parallel Sudoku Solving Using OpenMP

Luke Vandecasteele[1] and Matthew Trappett[1]

[1]University of Oregon

December 8, 2021

1 Introduction

With modern processors many serial implementations of traditional algorithms are extremely fast. However as data sizes increase and the need for these algorithms to stay fast is crucial, it is vital that algorithms evolve and begin to take parallel approaches to problem solving. A Sudoku solver is one such example, and with numerous techniques these solvers can oftentimes be very quick, even with serial implementations. However, what happens when these puzzles get larger, from 9x9 to 16x16 or even 25x25? Many of these solving techniques quickly become less and less effective, and their execution times become slower and slower. This is where we step in.

There are already many parallel algorithms for Sudoku Solvers, each with different approaches and uses of hardware such as GPU's. For this project, our goal was to develop a Parallel Sudoku Solver using OpenMP(1) by implementing our own parallel algorithm with the idea of practicing parallel programming, as well as improving our existing C/C++ skills to create a unique project.

Throughout the rest of the paper we hope to provide the motivation behind improving serial Sudoku solvers, a description of our version of a serial method and the tactics and algorithm being used, and finally, how we were able to effectively alter the serial algorithm to allow for proper concurrency in our parallel implementation, resulting in a significant increase in performance and speedup.

2 Description of Work

For this group project we ran into a few hurdles throughout the course of the term. One of these was a need to switch our original project idea (Gorila). Because of this, along with a few unfortunate technicalities regarding our new project (Sudoku Solver), our workload distribution become off-kilter from what one would expect from this sort of project.

For our first project idea, much of the work was conducted by Matthew Trappett, setting up libraries and working to provide a good foundation for beginning our project. The first project will be described in more detail in the next section.

When this failed, the new project idea consisted of the source code for the project (in C++) being written by Luke Vandecasteele with the execution times and data collection being done by Matthew. For the presentation, as well as this final report, the sections de-

scribing the Gorila algorithm were written/presented by Matthew, whereas the sections about the Sudoku Solver were written and presented by Luke. The idea behind this section was to provide a more complete picture of the total amount of work being completed by both group members.

3 Gorila

Our initial project aim was to recreate a parallel-distributed reinforcement learning framework called Gorila (General Reinforcement Learning Architecture). This was an ambitious project, but one we felt would have been possible. Implementing Gorila would have been a good balance of our programming strengths while helping us to improve.

3.1 Gorila Description

Gorila separates out the pieces of a reinforcement learning agent for asynchronous computation. Figure 1 shows the organizational breakdown of each piece of the Gorila framework. To implement the parameter server we would use pyCUDA or pyOpenCL and apply the gradients to the weights on GPUs. For passing the information between the actors, learners, replay memory and parameter servers we would use mpi4py. While this problem seems straightforward from a parallel computing viewpoint, it would have taught us a lot about parallel programming. We would have gotten significant practice with MPI since much of the parallelism is achieved by information passing from one piece to another.

3.2 Roadblock

Once we were familiar with Talapas, we started working on getting libraries installed. There were a few key libraries we needed: mpi4py, pyCUDA or pyOpenCL, and openAI GYM with Atari. Unfortunately, after much trial and error we were unable to get any of the libraries installed on Talapas. One exception was OpenAI Gym. However, the authors had decided a few months prior to remove the Atari game ROMs and files. Without the files, we would have been unable to compare our results with the original paper. As time progressed without solutions being found, we decided to switch to our Sudoku project to take advantage of the time remaining for the term.

4 Parallel Sudoku

This brings us to our second project idea and the main focus of this report: a Parallel Sudoku Solver using OpenMP(1).

For this project, we first had to find a suitable serial implementation to use in order to develop our parallel algorithm. Our serial implementation is based off of the overall structure of a Sudoku Solver created by Dr. Michal Young at the University of Oregon(3). This algorithm is composed of two steps: a pair of different tactics by which to fill in values of the puzzle and eliminate possible candidates for empty squares, and a recursive guess and check. The pair of different tactics were also used by another parallel Sudoku solver created by Sruthi Sankar(4), which provided the original inspiration for our own parallel algorithm. Our goal will to create a parallel algorithm based off of the guess and check method for solving a Sudoku puzzle.

4.1 Motivation

As previously hinted at, serial implementations of serial Sudoku solvers are already very fast. With a rather straightforward implementation, these sorts of solvers can solve any 9x9 puzzle you throw at it in under a second as seen in the paper by Armando B. Matos(5).

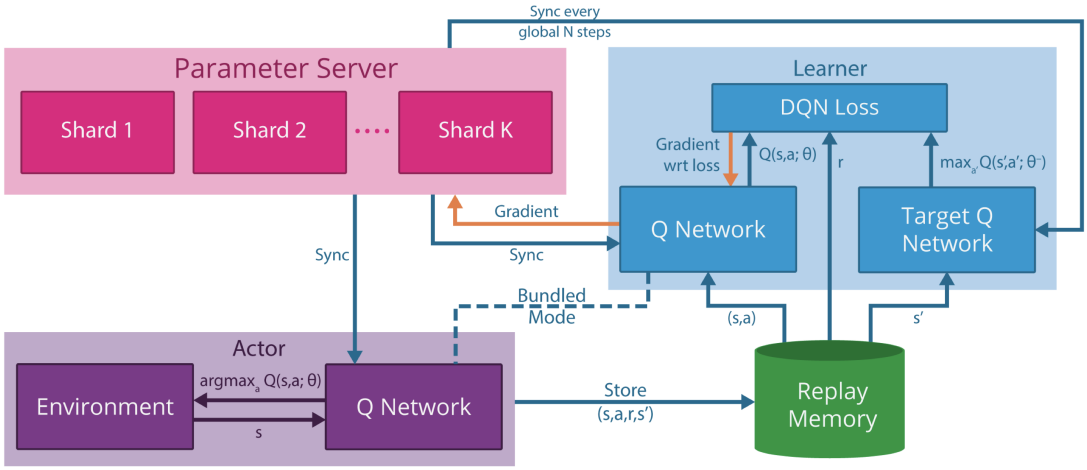


Figure 1: The actor takes actions using the current policy. Those actions are stored in a shared memory called the replay memory. The learner draws from the replay memory to calculate the loss and gradients. The gradients are calculated with by the parameter server. The parameter server updates the policies for the actors and learners every global step. A target policy lags N global steps from the synced policy of the learners and actors. The goal is to train the target policies to a desired convergence. (2)

Although the details our our serial implementation will come later, we can first look at the potential performance boost we are looking for with our parallel algorithm. First, we can examine the execution times for a serial Sudoku solver using two basic solving tactics and a recursive guess and check:

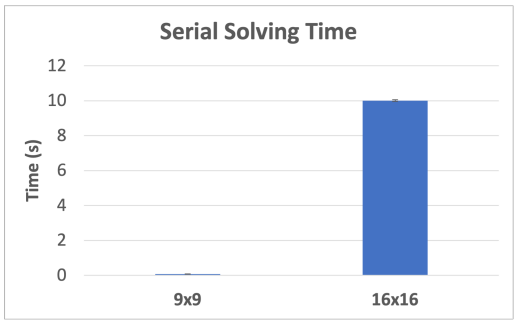


Figure 2

As we can see, our 9x9 implementation is very quick, solving the most difficult puzzles in under 0.1 seconds. However, for larger 16x16 puzzles, our serial method already becomes very slow with an execution time of almost 10 seconds. This where there is room for improvement.

4.2 Serial Method: Tactics and Guess and Check

First, we must discuss potential algorithms that could be used to solve a Sudoku puzzle. The simplest of these is a brute force algorithm which, given enough time, could solve any Sudoku puzzle. However, there are approximately $6.67 * 10^{21}$ different 9x9 puzzles(6) and an estimated $5.9584 * 10^{98}$ different 16x16 puzzles(7). This means that a brute force algorithm is not feasible.

There are also many other solving tactics that could be used. These include Simulated Annealing(4), Linear system approach(4), X-wing(3), Swordfish(3), and Forcing Chains(3). However, all these algorithms have weakness. Some of the methods are unable to solve difficult 9x9 puzzles even in combination with other tactics, and others have little work that can be made parallel.

Therefore, for our algorithm we de-

cided to use a combination of two tactics: elimination and lone ranger(4). These tactics by themselves can solve most 9x9 puzzles rated easy or medium. If we alternate these tactics with a recursive guess and check than we can solve any Sudooku puzzle.

4.2.1 Tactics

First, we have our tactics elimination and lone ranger. Both are simple enough. The elimination tactic checks for each row, column and nonet (the smaller $\sqrt{n}$ by $\sqrt{n}$ squares inside the puzzle) if there is an empty square with only one possible candidate:

1	2	3	4,7	5,8,9	6	4,7	5,9	5,9

Figure 3: Elimination example for a row of a puzzle. Since 6 appears only once as a candidate for all the empty squares in the row, it must be the value for that empty square.(4)

The lone ranger tactic checks for each row, column and nonet if there is any empty square in which a candidate appears only once, even if it is not the only candidate for a particular empty square. However, since the candidate only appears once in the row, column or nonet, than it must be the value for that square:

1	2	3	4,7	5,8,9	6	4,7	5,9	5,9

Figure 4: Lone Ranger example for a row of a puzzle. Since 8 appears only once as a candidate for any square in the row, it must be the value for that empty square.(4)

These two tactics are extremely fast, and the primary method by which values will be filled into empty squares in the puzzle. As previously stated, these two methods can solve all 9x9 rated easy, and some puzzles rated medium. However, for a majority of 16x16 puzzles, and all puzzles rated hard or extremely hard, these two methods will only be successful in filling in a few values in the overall puzzle.

4.2.2 Guess and Check

This brings us to guess and check. After apply our tactics, elimination and lone ranger, our puzzle may not be solved. Therefore, we need to find a new way to make progress in solving our puzzle.

For our recursive guess and check we first need to determine a good way of choosing a puzzle to make a guess for. Clearly, we first should be guessing only for tiles that are currently empty. Amongst our empty tiles, we should first make sure that our guess only comes from the list of possible candidates that could occupy the value of that empty square in the current state of the puzzle. This way we are only making guesses that, at this moment, could be possible values to solve the puzzle. Next, we want to choose the square in the puzzle that is the minimum square. The minimum square is an empty square in the puzzle that has the least amount of possible candidate values. This ensures that each time we make a guess, we have a higher likelihood of guessing correctly, since there are less possible values to guess from.

Now that we have our empty square to guess for, and our list of candidate guesses for that particular square we begin the recursive part. For each candidate value for the tile we first save the current state of the puzzle, set the value of the empty tile to one of the candidate values, and then recursively call our elimination and lone ranger tactics. If we solve the puzzle, then we are finished. If the puzzle is not valid, then we make a new guess. If the puzzle is not complete and valid, then we repeat the process.

4.2.3 Algorithm

The overall algorithm for our serial implementation is given below. First, we continue applying elimination and lone ranger tactics to the puzzle until we no longer make progress. Next, we check if the puzzle is still valid and if the puzzle is complete (this is important for the recursive guess and guess). Next, we step through the guess and check in the manner described in the previous section, recursively calling the solve function until we have a completed and valid puzzle to return up the recursive stack.

Algorithm 1: Serial Solver

```
1  def solve():
2      progress = 1;
3      while progress do
4          progress = elimination()
5          loneRanger()
6      if puzzle is not valid then
7          return 0
8      else if puzzle is complete then
9          return 1
10     else
11         minTile =
             puzzle.getMinTile()
12         save = puzzle.saveState()
13         n =
             minTile.candidates.len()
14         for i = 0 to n do
15             puzzle[minTile.x][minTile.y]
                 =
                 minTile.candidates[i]
16             if solve() then
17                 return 1
18             else
19                 puzzle.restore(save)
20 ;
```

4.3 Parallel Method

In order to improve our serial algorithm we first need to look for areas of potential speedup. In our algorithm, the primary cause of a slow execution time is due to the amount of wrong guesses we may make during the recursive guess and check. Each time a wrong guess is made, the algorithm must return up the recursive stack and make another guess for the puzzle.

4.3.1 General Idea

For the puzzles that cannot be solved by only our elimination and lone ranger tactics, there is only one possible combination of values that will result in a complete and valid puzzle. Thus, we can think of our algorithm as a binary tree. Each time we are forced to make a guess, there are n different branches of the same puzzle that are created where each permutation of the puzzle has a different candidate value in place of an empty square. Therefore, since our puzzle only has one possible solution, only one branch of our binary tree is going to contain the correct path of guesses in order to reach a valid complete solution for the puzzle.

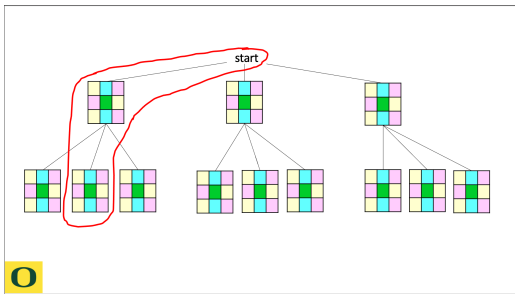


Figure 5: One path of guesses to solve the puzzle

By extension, this means that to create a parallel algorithm based off of this guess and check method, we need to find a way to effectively explore different paths of the puzzle and run our tactics,

elimination and lone ranger, for different permutations of the puzzle, each containing a different path of guesses.

4.3.2 Naive Approach

Taking a naive approach, one could simply parallelize the "for" loop in our serial algorithm such that each thread recursively calls solve() for different candidate values of the same empty square. However, there are many issues with this approach. First is the problem of recursion. In this method, each thread will recursively be calling the same solve() function. Therefore, next time each thread reaches the same "for" loop, it will encounter another parallel region and create more threads. The overhead of this thread creation model will significantly slow down our overall algorithm, potentially making it even slower than our original serial implementation. This is similar to what happens with a recursive Fibonacci sequence that is implemented with a naive parallel approach. Secondly, this method results in a race condition wherein each thread is altering the same puzzle while applying the two tactics.

4.3.3 Solution

We can solve these two issues by first, creating copies of the same puzzle each time we need to make a guess, and second, using a stack to remove recursion and instead use an iterative approach. Each time we make a guess, we can create n different puzzles for each of our n different candidate values. Next, for each copied puzzle, we fill in a different candidate value for the same empty square. Then we put each puzzle in the stack so that the next time through, we can pop of a puzzle in the stack, apply our tactics, and then repeat the same checks if the puzzle is valid and complete.

One problem with this method is the slowness of copying the entire puzzle, as well as the memory required to store an entire copy of the puzzle in the stack. As we have previously discussed, we can think of our guesses to solve a puzzle as making our way through the binary tree to find the one correct path. Since we know that only one path of guesses will lead to a solved puzzle, we do not have to store the entire puzzle in our stack. Instead, we can simply store the path of guesses (in an array) that may or may not solve the puzzle. Then, at each iteration, we pop a path off the stack, fill in the guesses into their correct squares, and finally apply our two tactics to solve the puzzle.

4.3.4 Algorithm

The overall algorithm is as follows. First, we apply our two tactics to make as much progress as we can before starting our guess and check. Next, so that each thread has some work to start, we initialize our stack with some starting paths to solve. For this, the number of minimum tiles chosen is equal to the number of threads that are going to be solving the puzzle. Therefore, the number of paths in the starting stack will be at least $2 * number of threads$. Next we start our loop. First, each thread pops a path off the stack. Then, each thread generates their own puzzle copy and fills in this puzzle with the path values from the stack. Next, each thread applies the two tactics. Finally, each thread repeats the same series of checks on the current state of the puzzle from our serial algorithm. If the puzzle is not valid, then we pop another path off of the stack. If the puzzle is complete then cancel all threads and return the completed puzzle. If the puzzle is valid and not complete then find the minimum tile for this permutation of the puzzle and, for each candidate value in the minimum tile, append the new value to the end of the current path and store this new path on the stack. Once done,

the thread starts again by popping a new path off the stack.

4.3.5 Cancelling Threads

One very important part of this algorithm is the cancelling of threads. When the puzzle is solved, we need the thread with the correct path to cancel all other threads. At first, this was simply implemented by putting a cancellation point at the end of the "while" loop. However, after reading an article by Jaka's Corner(8) where and how many cancellation points are in a loop can greatly effect the execution time of an algorithm. As such, cancellation points have been placed throughout the while loop at various points: one at the start and beginning, one before popping a path off the stack, and one before starting the two tactics. These cancellation points were found to have produced the best execution times for this algorithm, and by quite a significant margin when compared to only using only one cancellation point at the end of the loop.

Algorithm 2: Parallel Solver

```
1  def solve():
2      puzzle.applyTactics();
3      Stack s;
       // initialize stack
4      for i to numThreads do
5          minTile =
            puzzle.getMinTile(i)
6          n =
            minTile.candidates.len()
           for i to n do
7              s.push(
                minTile.candidates[i])
       // parallel region
8      while s is not empty do
9          cancellation point;
           // make copy of puzzle
10         myPuzzle = puzzle.copy();
11         cancellation point;
           // get path off stack
12         path = s.pop();
           // add path to our
              puzzle
13         for i to path.len() do
14             tile = path[i];
               myPuzzle[tile.x][tile.y]
               = tile;
15         cancellation point;
           // apply tactics and
              check puzzle state
16         myPuzzle.applyTactics();
17         if puzzle is not valid then
18             continue;
19         else if puzzle is complete
            then
20             puzzle.restore(myPuzzle)
               cancel all threads
21         else
22             minTile =
                puzzle.getMinTile();
23             n =
                minTile.candidates.len();

24             for i = 0 to n do
25                 path.append(
                    minTile.candidates[i])
                   s.push(path)
26         cancellation point;
```

5 Results

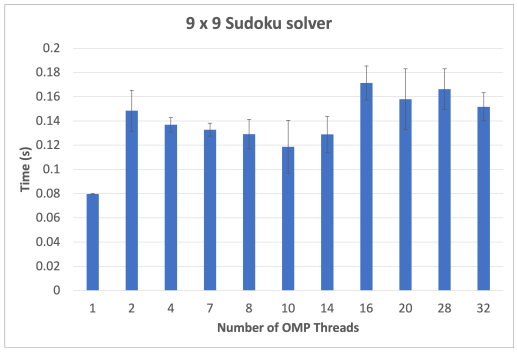
Our results were generated by running our code on a short partition of the Talapas HPC cluster at University Due to our code implementation we were only able to use up to 32 OMP threads but we thoroughly tested our parallel versions between 2 and 32 threads. When runs used 14 or less threads we only scheduled 14 threads on Talapas. For our runs we ran with OMPPLACES='cores' to prevent the scheduler increasing run time by moving threads around. We also had OMPPROCBIND='spread' to spread the threads around cores near each other. We ran a test for the given number of threads 6 times from which we took the average and calculated standard deviations. Charts show the mean with

error bars being a single plus and minus standard deviation.

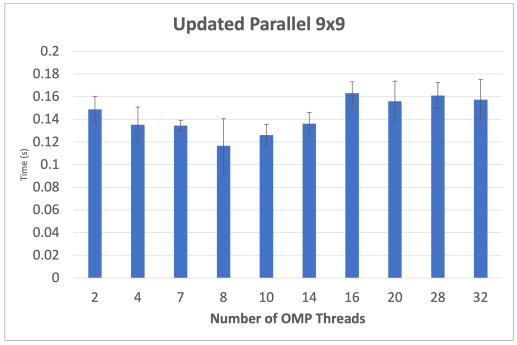
```
Dimension: 16
Puzzle:
        0 5 2 7 . 6 . . 4 . 3 . . . 9 .
        9 e . . b 2 . . . 6 . 1 . 5 3 0
        8 . . . 0 . . 4 9 . 5 . . 6 7 a
        . . . . . . . . e . . 7 . . c .
        . . . e 1 c . 2 3 4 b 6 . . . .
        . a 3 . . . . . . 2 . . 8 4 . .
        . . . . . 3 . 9 d c . . . 7 . 5
        1 f 6 . . e . . . . . 9 . . . .
        . . 9 f . . . a . . . . c e 8 .
        . 4 . . 9 . . . . . . 0 6 . . .
        . . 1 b . 8 7 . . 9 . . . . . .
        . . . 5 c . . f . . 1 d . . b 2
        d . f . . . 6 c . . . e . . . .
        6 . . 4 . d . . . 8 . . . . 1 3
        . . e . . 4 9 . . . . 3 . . d .
        . 7 . 2 f . . 1 . . . 4 . c . .
Checking valid puzzle...success
Parallel method...done
Time to solve: 2.3539
Dimension: 16
Puzzle:
        0 5 2 7 8 6 a d 4 f 3 c b 1 9 e
        9 e 4 a b 2 c 7 8 6 d 1 f 5 3 0
        8 3 c 1 0 f e 4 9 b 5 2 d 6 7 a
        f d b 6 5 9 1 3 e 0 a 7 2 8 c 4
        7 9 5 e 1 c 8 2 3 4 b 6 0 d a f
        b a 3 d 7 5 f 6 1 2 0 9 8 4 e c
        4 2 0 8 a 3 b 9 d c e f 1 7 6 5
        1 f 6 c d e 4 0 a 5 7 8 9 3 2 b
        2 0 9 f 6 1 d a 5 3 4 b c e 8 7
        e 4 d 3 9 b 2 5 c 7 8 0 6 a f 1
        c 6 1 b 4 8 7 e 2 9 f a 3 0 5 d
        a 8 7 5 c 0 3 f 6 e 1 d 4 9 b 2
        d 1 f 9 3 7 6 c 0 a 2 e 5 b 4 8
        6 c a 4 e d 0 b f 8 9 5 7 2 1 3
        5 b e 0 2 4 9 8 7 1 c 3 a f d 6
        3 7 8 2 f a 5 1 b d 6 4 e c 0 9
Checking valid puzzle...success
```

Figure 6: An example output from our algorithm. A puzzle is written in a text file and read in by a parser. Tests are performed to validate the puzzle. Th time to solved is given with the answer puzzle displayed after. A final check is performed on the answer puzzle.

An example output of the Sudoku solver is given in figure 6. The serial version was already very fast when solving 9x9 puzzles. The average speed was 79.7 ms while the fastest speed was 77.948 ms. All other parallel runs never completed faster except for after updating the parallel version. Our fastest solve for 9x9 Sudoku was 74.693 ms using 8 threads. A comparison of the best time is shown in figure 9a. However the average was



(a) Comparison of times between serial (1 thread) and parallel (2+ threads) versions of solving 9x9 Sudoku puzzles.



(b) Comparison of time to solve 9x9 using updated parallel algorithm.

Figure 7: Charts for average time to solve 9x9 Sudoku puzzles. Error bars are single plus and minus standard deviation about the mean. For each number of threads runs were repeated 6 times and then the average was taken.

that serial was faster. Figure 7 compares the means of all runs for solving 9x9 puzzles. The parallel version never differ much after our updates, however 8 threads on our updated version have the best average performance out of the parallel version which matches with the best time for 9x9 solving. The serial average is the best. We believe the reason the parallel versions were so slow was due to the amount of overhead that was created with each additional thread. The serial was so fast that any extra overhead slowed down the total computation speed significantly.

However, we were most excited to speed up the 16x16 solving time and we

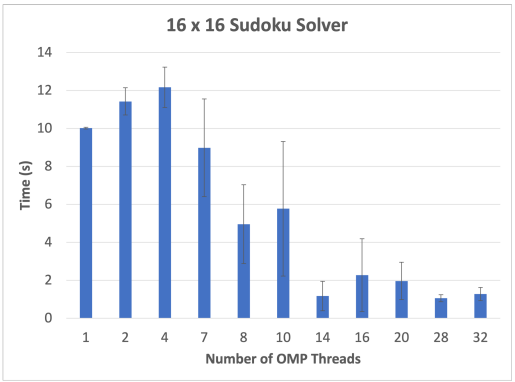
definitely succeeded. Figure 8 compares the mean times for serial and both parallel versions. Error bars depict plus and minus a single standard deviation. All runs were completed using the same environment variables and repetitions as with the 9x9 data.

Figure 8a shows how much using more threads decreases the solve time of a 16x16 puzzle. When using 2 and 4 threads the time does actually increase as the amount of work has not been split up enough to improve performance and negate the amount of overhead created. Beyond using 7 threads and more decreases the overall all time by finally spreading enough work out amongst the threads to improve performance despite the overhead created. When using 14 and more threads the run times become the best. This trend is also seen in our updated parallel version in figure 8b but only using 4 threads seems to be the only one that averages worse than the serial version.

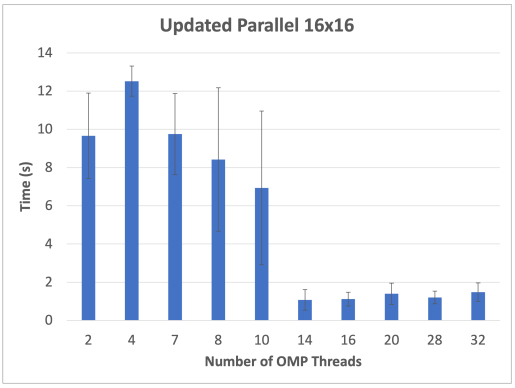
Using 2 threads averages better but has a large variance between times and so is less consistent. When using 10 threads and less the variance between run times appears to be quite large, compared to using 14 and more threads. That variance may be attributed to using only a single socket on a processor and thus the run-time system may schedule another job on the other socket possibly interfering with our times.

Solving times are more centered about the average when using 14 or more threads. Our updated version has even smaller standard deviations and have times more similar to each other than our first implementation. By comparing the time spread shows how much improved our updated algorithm is compared to our first attempt.

Figure 9 compares the best times out of all of our runs. The configurations for the best 9x9 puzzle solving time used

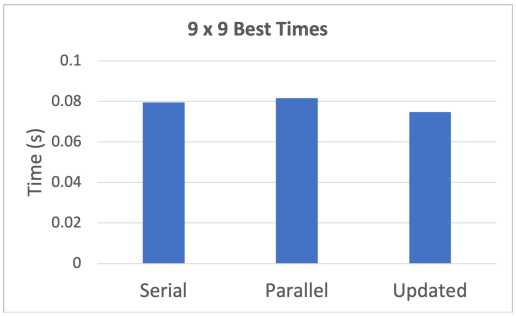


(a) Comparison of times between serial (1 thread) and parallel (2+ threads) versions of solving 16x16 Sudoku puzzles.

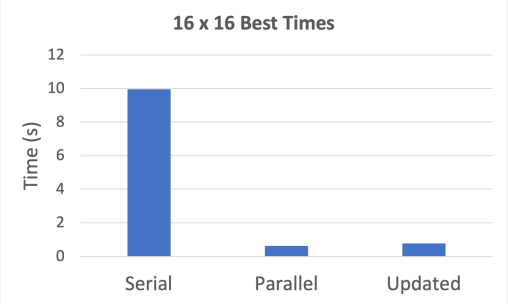


(b) Comparison of time to solve 16x16 using updated parallel algorithm.

Figure 8: Charts for average time to solve 16x16 Sudoku puzzles. Error bars are single plus and minus standard deviation about the mean. For each number of threads runs were repeated 6 times and then the average was taken. Significant decrease in computation when using 14 and more threads.



(a) Comparison of best 9x9 solving times. Parallel used 10 threads and updated used 8 threads. Practically no improvement made when using a parallel solver compared to the serial method.



(b) Comparison of best times for solving 16x16 puzzles. Both parallel and updated used 14 threads for the best time. Our parallel method saw a 15x increase in performance.

Figure 9: Best times out of all serial and parallel runs for solving Sudoku puzzles.

8 threads after updates. Before the updates, the 10 threaded configuration was the best time. For the best 16x16 puzzle solving time configuration, 14 threads was the best for our original and updated parallel versions. One thing to highlight is the difference in scale between 9x9 and 16x16 puzzle solving times.

Conclusions

As we can see by applying a parallel thought process to a serial Sudoku solver we can get a massive performance boost. What is very interesting about a project like this is that there are numerous different methods and algorithms one could apply to the problem and come up with very different solutions. Although this project was ultimately successful, and taught us a lot about parallel programming through our implementation and work, there is still room for improvement and different ways of building off of this project.

The first of these would be to try and apply other API's such as MPI to our same algorithm. Furthermore, with a parallel algorithm such as ours, one could easily convert the algorithm to be done on a GPU, since exploring the binary tree is a sort of SIMD architecture by which the data is the various permutations of puzzles in the tree, and the instruction is our two tactics being applied to each permutation of the puzzle. With these different methods it may be possible for this algorithm to solve puzzles even larger than 16x16, which is something our current implementation was not able to do.

Ultimately, we feel as though this project was significantly complex and able to give us invaluable experience with parallel programming, along with a taste of some of the difficulties and challenges of finding parallel work in an algorithm, even going as far as to alter algorithms in order to increase the amount of parallel work. This project also showed us the difficulties that can arise when working as a group to find suitable areas of focus to research in and apply parallel methods.

Acknowledgements

This work benefited from access to the University of Oregon high performance computing cluster, Talapas.

References

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[2] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[3] M. Young, "Cis 211, spring 2020introduction to computer science 2." [Online]. Available: https://classes.cs.uoregon.edu/20S/cis211/projects/sudoku.php

[4] S. Sankar, "Parallelized sudoku solving ... - university at buffalo." [Online]. Available: https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Sankar-Spring-2014-CSE633.pdf

[5] A. Matos, "The most difficult sudoku puzzles are quickly solved by a straightforward depth-first search algorithm," Sep 2016. [Online]. Available: https://www.dcc.fc.up.pt/~acm/sudoku.pdf

[6] B. Felgenhauer and F. Jarvis, "Enumerating possible sudoku grids," Jun 2005. [Online]. Available: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[7] "Number of possible 16x16 sudoku grids? : General." [Online]. Available: http://forum.enjoysudoku.com/number-of-possible-16x16-sudoku-grids-t37053.html

[8] Jaka. [Online]. Available: http://jakascorner.com/blog/2016/08/omp-cancel.html