15-418 Final Report

andrewID: waltert

Summary

We parallelized a brute-force Sudoku solver algorithm. We have implemented both serial and parallel versions of the same algorithm. Running the solver on the machines in the Gates cluster shows very good speedup across varying number of cores used. The solver also works on larger square Sudoku boards, with grid dimensions greater than the standard 9x9 grid.

Background

Sudoku is a combinatorial puzzle game where the goal is to fill a 9x9 grid with numbers such that each row, column, and each of 9 3x3 subgrids contains each number 1 through 9 only once. At the start of the game the player starts with a partially filled grid, where the selection and amount of squares filled typically affects the difficulty of that particular puzzle. Sudoku puzzles are typically 9x9 grids with 3x3 subgrids, but also come in many variants. Smaller and larger grid sizes are possible, as well as different shapes. Variants also include extra constraints, which can limit the ways numbers can be filled.

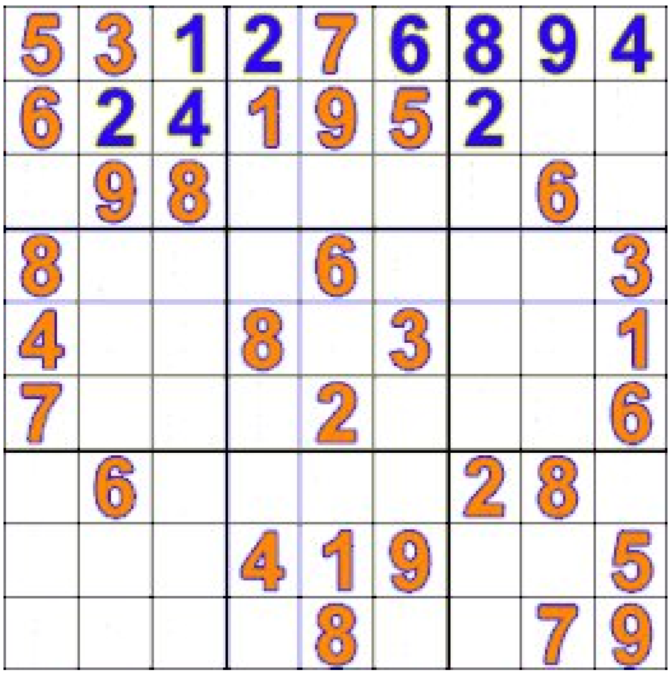
Our algorithm's inputs are Sudoku grids containing some number of cells filled in already. These inputs are simply represented as text files containing rows of the filled in numbers, and 0 for representing an unfilled cell. The algorithm's output is a printed solved Sudoku grid, or no solution if the original input was unsolvable. The computation time needed to find the solution is also output. The key data structures used were Matrix objects and a single job queue. A Matrix object is represented as a 2D grid of numbers, representing a Sudoku grid’s current state of some cells being filled in and other cells being empty. The purpose of the job queue will be elaborated later, but it is implemented as a simple linked list, and contains Matrix objects. The Matrix object supports reading and writing operations to any of its values, as well as a legality checking function, which checks if placing a certain value in a grid’s cell is allowed. This function simply checks if the value is already present in the cell’s row, column, and 3x3 subgrid. If the considered value is already present in any of these, then placing the value in the cell is illegal, as per Sudoku rules. The job queue supports an insert operation, which adds a Matrix object the queue’s tail, and a remove operation, which retrieves and removes a Matrix object from its head.

Sudoku solvers are programs that find solutions to Sudoku puzzles, and typically do so through brute force search. Given a start grid, the solver will attempt every possible combination of numbers for the grid for the unfilled values. In particular, brute force solvers use backtracking algorithms, where branches of possible solutions are explored completely before moving on to the next branch. The computationally expensive part of is the whole search itself, since we are brute forcing and searching for a solution over all possible combinations of legal values filled in. The number of possible combinations is exponentially large in the number of unfilled cells, so parallelism gives the perfect opportunity to speed up this search. Not all search paths are dependent on each other, so different patterns can be searched for in parallel.

Our solver algorithm uses brute force search, where given an input of a partially filled board, it will visit the unfilled cells in some order (in our algorithm, the order was left to right and top to bottom, starting with the top-left cell of the grid), fill in the cells with legal numbers sequentially, and backtrack when a number is found to be invalid.

At first, our serial algorithm used a pure backtracking algorithm to implement brute force search. Backtracking is a type of depth first search, because it explores one branch of a possible solution completely, before backtracking and trying another branch.

Here is a brief explanation of what a backtracking algorithm would do. On visiting the first unfilled cell, it would attempt to place a “1” value there. If that value is legal, meaning that there is no other “1” in the cell’s same row, column, and subgrid, then that value is placed. Otherwise, the value is incremented, and the new value is tested for legality. After a legal value as been found and placed, this whole process is repeated for the next unfilled cell in order. If at some point all 9 digits are illegal for a cell, that algorithm backtracks by erasing this cell and then going back to the previously filled cell. At this cell, the value is incremented and tested for legality, until a new legal value can be placed. Then the algorithm moves on to the next unfilled cell. This general process is repeated until a legal value is placed for all unfilled cells of the grid. At this point, we have a solution to the puzzle (assuming that the input was a valid board).



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

Figure 1: A snapshot of the grid with the backtracking algorithm running.

For example in Figure 1, the values in orange are given and fixed by the input, and the values in blue are placed by the currently running backtracking algorithm. The algorithm is currently testing a value of “2” in the upper-right subgrid, but this value would not be legal since a “2” already exists in the same row and same column. Thus the algorithm would increment the value to “3” and place that since it is a legal value. It would then move on to the next unfilled cell one position to the right. Hypothetically, if none of the values from 1 to 9 were legal for the cell that 3 was just placed in, the algorithm would erase the cell, backtrack to the cell in the upper left subgrid containing the blue 4, and place a new next legal value in that cell, before moving on back to the next unfilled cell.

However, we noticed early on that using a pure depth first search for the brute force algorithm would be sufficient for a serial algorithm, but hard to parallelize effectively. This is because there is no clear distribution of work among workers. Although the brute force algorithm explores many branches, these branches are of varying length, so finding a fair distribution of work would be challenging. Moreover, these branches would begin at varying depths in overall computation tree, so finding all branches at the very beginning of the algorithm would be challenging. To overcome this challenge, we implemented a brute force solution that uses a combination of breath first search and depth first search.

In this solution, the algorithm first initializes a pool of jobs and adds them to the job queue. The number of jobs initialized is a small number that is at least the number of available workers. In our algorithm we just set this number equal to the side length of the a grid, which is 9 for the 9x9 input board. An initial job represented by just the initial grid plus 1 or more of the initial empty cells filled in. Given the initial input grid, the algorithm will run breadth first search on it to generate the initial jobs. For example, the algorithm will visit the first unfilled cell, and try to place a value of “1” there. If it is legal, the updated grid is added to the job queue. Either way, the algorithm proceeds by staying at the same cell, trying the next incremented value, and only adding the updated grid if the newly placed value is legal. If all the values 1 through 9 are tested for this cell and the job queue is still too small, then the algorithm removes the oldest job in the queue, takes that job’s grid, moves on to the next unfilled cell, and tests legal values there. It will add to the queue jobs with grids that have this new cell legally filled. This is repeated until the the number of initial jobs required is met.

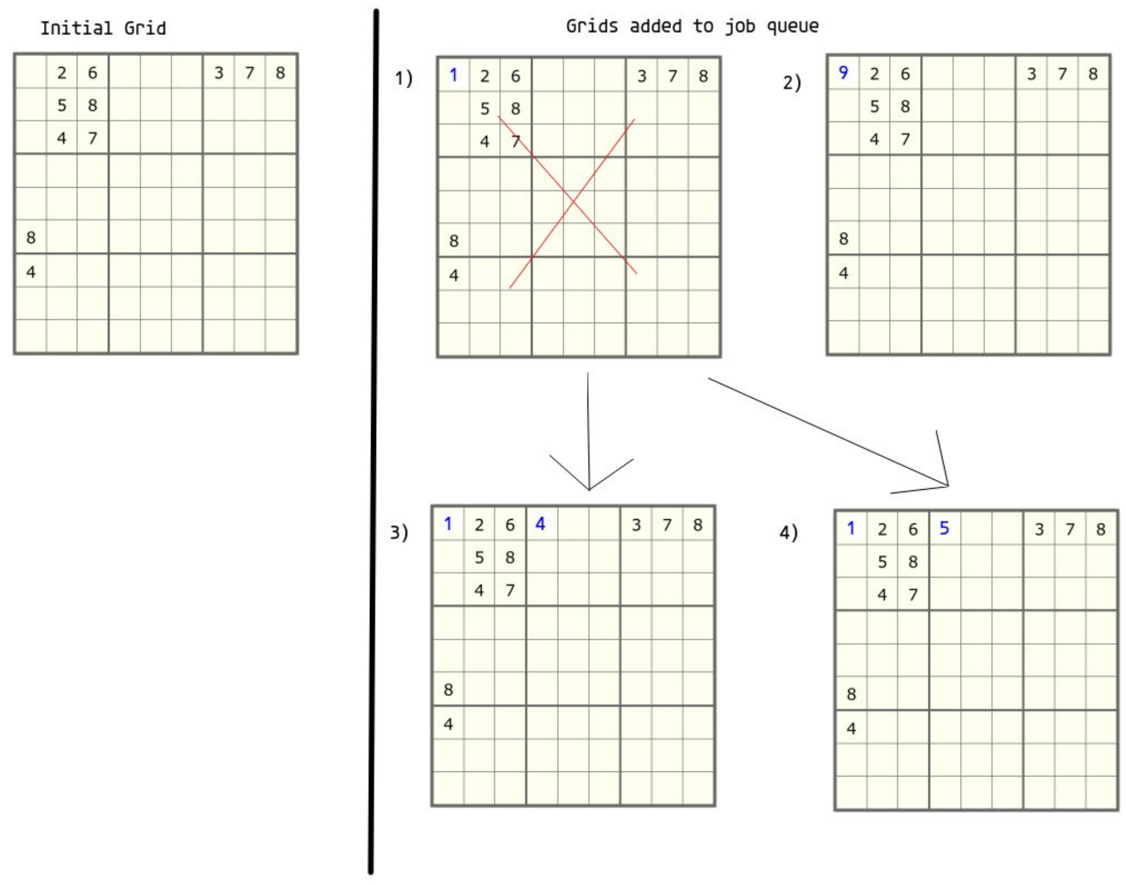


Figure 2: Algorithm’s initial BFS search and first few initial jobs added

For example in Figure 2, the algorithm will run BFS on the initial grid and add Grid 1 then Grid 2 into the queue because these have legal values for the first unfilled cell. But since there are no more values left for the first cell, the algorithm will remove Grid 1 from the queue, continue from its grid, move to the next unfilled cell, and tests legal values for this new cell. Grids 3 and then 4 will be added because they contain the next legal values for the new cell. When there are no more values left for this new cell, Grid 2 would be the next to be removed from the queue, and more jobs would be generated from its grid.

After this queue of initial jobs is initialized to a sufficient number, the algorithm will then remove jobs from the queue and assign them to available workers. This is where the parallel region begins, and ends only when a solution is found, or when all jobs are finished processing. Each job is processed by a worker who uses the depth first search backtracking algorithm describing earlier. Here, the backtracking algorithm begins from the job's corresponding grid, which will have the first or first few free cells filled out. Here there is opportunity for parallelism, since each initial job is independent and can be done in parallel across workers. These initial jobs are independent because they all have their latest filled cell filled by different values, so the jobs all resemble different branches of computation. These jobs are also relatively close in length, since they all consist of grids having only the first few free cells filled in. Thus the distribution of work among workers is pretty fair. The algorithm uses task-parallelism rather than data-parallelism, since tasks of different grid states are being distributed in parallel among workers. We found that as long as the number of initial jobs produced was greater than the number of available workers, the parallelism was pretty high, as initially no worker was left without an assigned job.

Approach

Our solver algorithm is written in C++, and uses the OpenMP API to implement multi-threaded and shared address space parallelism. We targeted our algorithm for the Gates cluster machines, which have 8 cores with no hyperthreading. Our testing was done by measuring speedups when using between 1 to 8 threads, as set by OpenMP. As a reminder, our algorithm has two main stages. In the first stage, it takes an input of a partially solved Sudoku grid, runs breath first search to find a small number of beginning states for the grid, and then adds these states as jobs into a shared job queue. In the second stage, the algorithm removes jobs in queue and assigns them to workers who use the depth first search backtracking algorithm to find a solution. The second stage takes up the bulk of the computation, but has potential for parallelism since the initial jobs in the queue are all independent of one another.

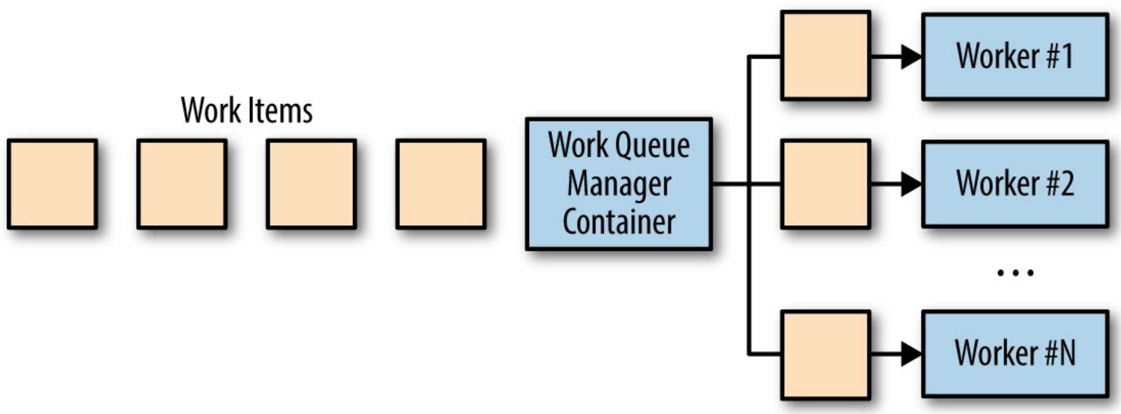


Figure 3: Outline of our algorithm's second stage, where the queue of work is distributed among several workers, who can work in parallel.

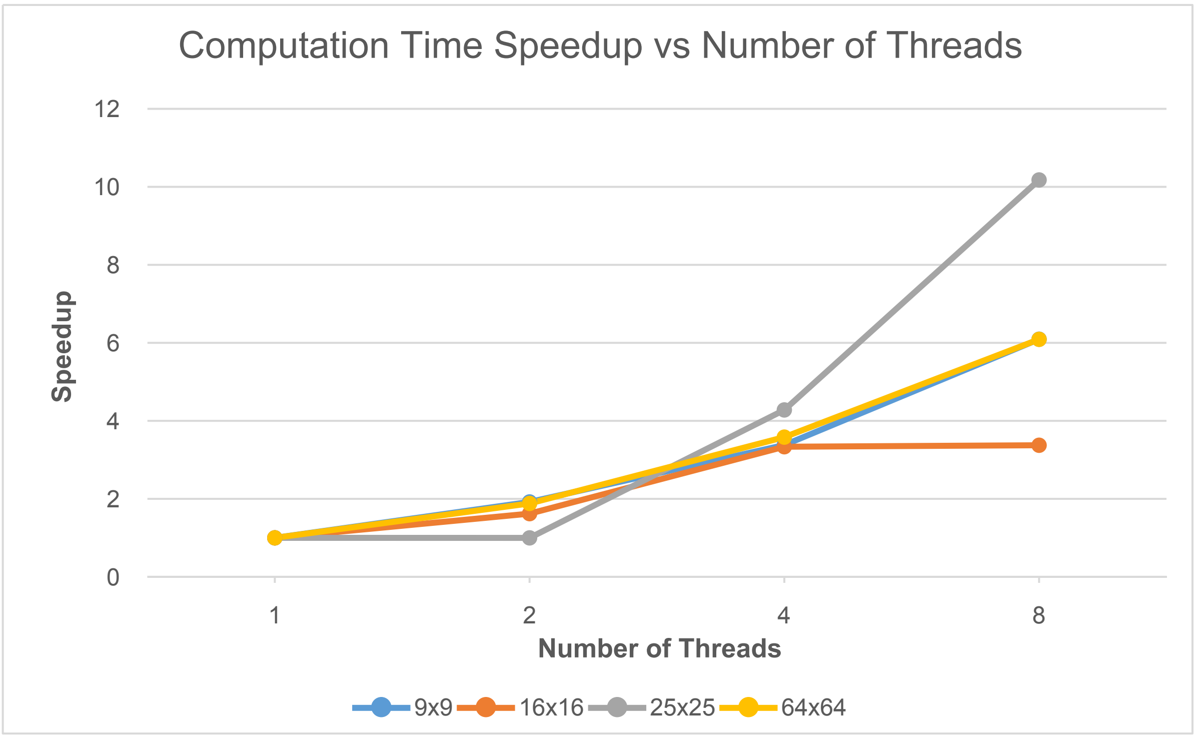
To parallelize the second stage, we used the fundamental OpenMP parallel construct, "pragma omp parallel", to declare this stage to be executed by multiple threads. In this stage, any individual running thread can remove a job from the queue and process it using the backtracking algorithm. If the thread is done with its job and no solution has been found yet, it will attempt to remove and process another job from the queue. If a solution is found (by the current thread or any other thread), all threads will complete and exit out of stage, after processing their current job. To notify other threads that a solution has been found, in the parallel region we use a single shared boolean variable that indicates whether a solution has been found. Other variables such as the level of the depth first search and position of the grid were declared private to each thread, since each thread would be working independently on its assigned job. To ensure correctness when removing jobs from the queue, we use the "pragma omp criticial" synchronization directive to indicate that the queue's removal function should only be executed by one thread at time. Without it, more than one thread could simultaneously attempt to remove jobs from the queue, possibly resulting in an incorrectly linked queue. When running the algorithm on the Gates cluster machines, the number of threads that OpenMP uses will be set through program input. At this second stage, a thread in OpenMP will be mapped to an available single core in the physical machine. So in this parallel region, each core will be running at most one thread, which will attempt to process jobs one at a time from the queue.

My original serial algorithm for the Sudoku solver was a pure backtracking recursive depth first search through all the possible solutions of the grid. This original algorithm was adapted from starter code found here: https://www.geeksforgeeks.org/sudoku-backtracking-7/. Since this would not be easy to parallelize, we adapted the original algorithm into the current and final algorithm, which uses breadth first search to initialize a queue of jobs, before using backtracking. We also converted the recursive backtracking algorithm into an iterative version, because this was more efficient on memory. The final algorithm was much easier to parallelize, since the job queue that it initializes consists of independent and equal size jobs.

Results

To measure our results, we used the cycleTimer code provided in Assignment 1 to measure the computation times on various inputs. This code uses the cycle counter of the processor, so we use it to obtain precise time measurements for our algorithm. We start our timer after reading in the input and before running breadth first search to initialize the job queue. We end our timer after a solution has been found and all threads have completed their current job. This effectively measures only the time needed to compute the solution and ignores any time used to process the input or print the output solution.

We tested inputs of square grids of various sizes ranging from 9x9 to 64x64. For 9x9 grids, we only considered more difficult cases because many simpler cases solved too fast (within a tenth of a second) and were unable to give any meaningful speedups when increasing the amount of cores used. The inputs were obtained from various websites offering Sudoku puzzles. For the various inputs, we measured speedups in computation time obtained when using 2,4, and 8 threads, compared to when just using 1 thread. When testing with 1 thread, we are used parallelized version of algorithm but set the number of threads to 1 using “omp_set_num_threads()” function. We measured the speedups on input sizes of 9x9, 16x16, 25x25, and 64x64.



Results were successful in most cases. Results show that for most cases, the computation time speedup increases as more threads are used. There are some outliers, such as no further speedup for the 16x16 input when going from 4 to 8 threads, and even more than ideal speedup for the 25x25 input at 4 and 8 threads. I speculate that these occur because of the specific values of input fed, rather than because of the input size. Depending on the values and locations of unfilled cells in the input, the initial queue of jobs may contain very unbalanced jobs, which would cause bizarre speedups when running with different numbers of threads.

In the normal cases, I speculate that lack of perfect speedup is caused by synchronization. Only one thread can access the job queue at a time, so synchronization costs increase as the number of worker threads increase. In particular, at the beginning of the job processing stage, there are many threads attempting to acquire a job from the queue, so there is high contention. If the threads work on jobs of around the same size, then many threads will finish their job around the same time and there will once again be heavy contention among threads in accessing the queue for more jobs. There is not much communication required among threads since they each work on their own job, and only write to a shared boolean variable if their job contains the solution.

We separated timed the first stage of our algorithm which runs breadth first search to initiate the job queue. The time spent in this stage was negligible compared to the overall computation time of the algorithm, so we decided it was not necessary to separate the stages when measuring computation time. Our choice of only using a CPU was sound for the 9x9 input sizes, since the most difficult puzzles of that size took 10s of seconds to solve. However, some puzzles of larger input sizes took at least several minutes to solve. In these cases, adapting our algorithm to run use a GPU instead of a solely on a CPU for computation may have been a better choice.

References:

1. https://www.geeksforgeeks.org/sudoku-backtracking-7/
2. https://alitarhini.wordpress.com/2011/04/06/parallel-depth-first-sudoku-solver-algorithm/
3. https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
4. https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/ch10.html

Distribution of Work for Project

Walter Tan - 100%

This project was done solo, so I did all parts.

Project Video: https://drive.google.com/file/d/1wx8JZa-gGyuG5a8D7S2Ohr2uDS-Bjhj-/view?usp=sharing