Parallel Minesweeper Solver

Jack Wong, Nathan Park, Thomas Wang

Department of Computer Science and Electrical Engineering,
University of Maryland Baltimore County
May 18, 2023

Abstract:

Minesweeper is a simple game with a very simple premise and easier gameplay that can lead to complex calculations and thought processes.
We have implemented some AI concepts to improve solving the game, but even with a serial approach, we have a performance issue. It attempts to solve the game, but depending on the board state, it could take upwards of minutes to solve for 20+ unknown tiles. We wanted to improve this efficiency by implementing parallelization to increase efficiency and overall runtime to improve the gameplay, and be able to beat/win the game in mere seconds instead of minutes.

The focus of this experiment is to explore parallelization in conjunction with a game-solving AI algorithm to measure the performance boost of parallelization. Additionally, to demonstrate its potential in solving or creating future algorithms for other games.

Keywords:

Parallel Algorithm, Minesweeper, C++, OpenMP

Motivation:

Minesweeper is a simple game that most programmers can easily code in less than a day. What we really wanted to do was to create an algorithm that gives us a nearly 100% rate of success in winning the game, which we can only hope to achieve. This is unfeasible, due to the randomness of the game, but we can strive for a higher chance of success. Our team has a few AI students and we wanted to find a way to get results and predict moves to solve Minesweeper faster. The implementation of our algorithm is what piqued our interest as it deals with the theory of information, where the known data, whether correct or incorrect, has value in our further analysis (which was implemented as a heuristic approach). That is, if we know for a fact that a tile cannot be a bomb, that data has affected the neighboring known and unknown tiles' information creating a cascading effect of improved chances.

The objective of this project was to see how we can increase AI solving and move making strategies for future games especially in static, turn based games.

Methodology:

The basic rules of Minesweeper are very simple and intuitive. A random spot on the board is clicked and several tiles become uncovered, some blank and some numbered. The numbered tiles indicate that there are that many bombs around that tile. When a tile is suspected to be a bomb, we can mark the tile with a flag to safely cover it up. The goal is to safely navigate around the board until all the safe spots have been uncovered. Should we incorrectly expose a tile, a bomb is detonated and the game is over. To prevent this, we have implemented an algorithm that greatly improves the chances of winning. With the addition of parallel processing, we were also able to improve the speed of our algorithm.

Before we go on to discuss our algorithm, we must first go over the conditions that make the algorithm needed. When looking at the simple gameplay of Minesweeper, we determined that most of the decision making and strategy falls under 2 rules. The first rule is that if there is a numbered tile and the number of neighboring unknown tiles is exactly the same as that number tile, then the unknown tiles are bombs. In other words, if the number on a tile is the same as the number of neighboring unknown tiles, then we are 100% certain that those unknown tiles are bombs. The second rule is that if there is a numbered tile, then the number of neighboring flags and bombs around the numbered tile cannot exceed that number. Simply put, if the total number of neighboring flags and known bombs is the same as the number on a numbered tile, then any neighboring unknown tiles are guaranteed to be safe. If these 2 rules cannot be applied to a tile, we have run into a case where our algorithm will be used. That is a case in which an unknown tile is not absolute, meaning that the tile is neither 100% safe nor 100% a bomb.

If we cannot use rule 1 and 2, we will instead use our algorithm, rule 3. This is because there are certain cases that require more thinking and strategy to determine whether a tile is a bomb or a safe tile. Those cases are often a result of the random nature of the game. Rule 3 is the basis of the algorithm, we have implemented a “counting method”, which is basically a forward checking procedure with exploratory search. It is somewhat of a variation of arc consistency in AI. Essentially, we place a theoretical flag on an unknown tile to try a possible outcome, then subtract 1 from all the neighboring known tiles around the tile we just marked. If there is a case in which one of those neighboring tiles’ number is less than or equal to -1, then it is no longer a possibility. In other words, we have finished checking that outcome but there are negative non-zero numbers, such that there are more flags and bombs than there are supposed to be, resulting in a contradiction. We also check if there are any open spots, if not, there is no possibility either. On the other hand, if all the numbered tiles are greater than or equal to 0, then we can continue checking this possible game state.

Further improvements to rule 3 have been made using heuristics and back tracking. The counting algorithm is a good way to check if a game state is possible. Backtracking allows us to try multiple possibilities by using recursion to test a case, then backtrack to a previous game state to try another possibility, if we have encountered an impossible game state.

We can improve this further in the form of pruning. There are two forms for this: one for the safe assumption and one for the flag assumption. The safe assumption needs to make sure that if we do mark a tile as safe, the known tile must be in a state such that it does not break the rules of minesweeper. That is, when we assume a tile is safe, then the adjacent known tiles must still have their respective number of bombs around them. The flag assumption has the same concept except that we are assuming a flag. That is, we place flags on the tiles with the highest probability of being a bomb, such as the MCV (most constrained variable).

Furthermore, we parallelize rule 1 and rule 2 by dividing and conquering all the known tiles until the two rules are no longer applicable. Then we backtrack to a game state that can no longer be solved with rule 1 or 2. In that case we need to do exploratory searching, where we use parallelization and recursion to explore all the possibilities. This will look similar to a merge sort where if it ever needs to explore a safe or flag assumption, we can assign a process to explore that tree. Each process will divide up the problem and explore the rest of the tree. To do this, we will be using a master-slave processing schema. Once a stable game state is reached, it will add it to memory and then get further processed so we can clearly see which nodes are statistically the better choice to flag or open. The main processes will divide the work with other processes to find all the different outcomes of flagging or unflagging tiles. We will store these game states to find the probability of whether a tile needs to be flagged.

Now that we have all the game states, we can use a simple probabilistic approach of how many times a tile is flagged in all the possible game states, which can be approached using a probability model instead of random. This results in a table that gives us a probability for each tile based on how many times each tile has been flagged. Absolutes are determined by 100% and 0%, meaning we flag and open tiles, respectively. We can first go through the absolutes before trying the next highest tile. For example, if we were to have a 99% tile and a 5% tile, we would flag the 99% and run through the algorithm again, since a 99% bomb rate is.

One way to introduce parallelization to improve performance is to parallelize the algorithm for rule 1 and rule 2 by doing a simple divide and conquer parallelization. Each rule can be done independently of one another since it is a simple 2D array traversal, then a calculation of whether rule 1 or rule 2 applies at a specific index in the array. We can then use a simple shared memory allocation to gather the moves, return back to the main thread and further process and apply the game.

Another way to parallelize this is when rule 3 is applied. This is also where it is the most resource intensive. Since we have to do an exploratory search and fully expand a binary tree, this will require lots of memory and processing power. However, this can easily be divided into a binary search tree type of problem where the last level of traversal, when we find a game state, will communicate its game state to the main thread. This is the master-slave approach where the slave thread will communicate via shared memory and add the finished game state to that shared memory. Once a game state is added, it will then be further processed by the main thread, since rule 3 will give us the probabilities. But we still need to choose what the best next decision would be, whether to do all the absolute actions or finding the next best action to perform. The absolute actions can tell the main thread that there are multiple moves to make, but if there are no absolutes, it will only send one action to the main thread, so it can update the game state, then reprocess the game data with new information.

How we are testing the performance of our algorithm by using a timer from the C library to record the time from beginning of the problem to the end. We start when the solver does the first move and end when the board is completed. This was tested using 1000 trials, and the average of those trials were taken.

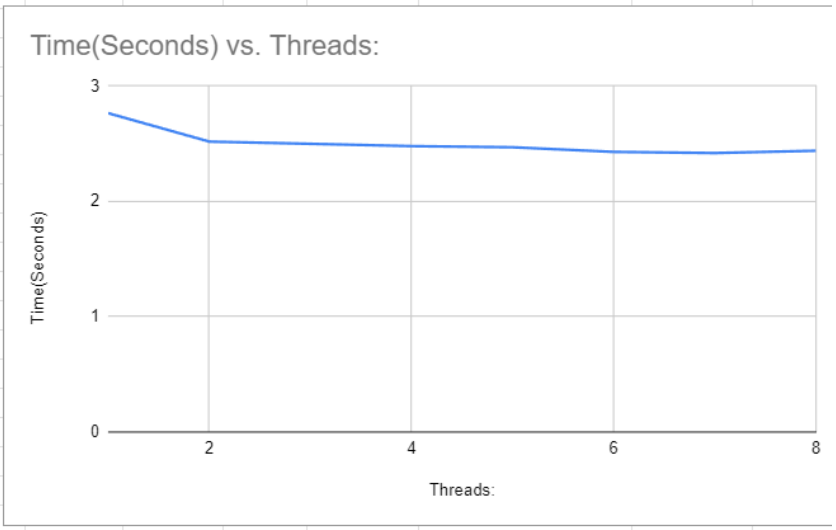
We found that rule 3 has a runtime of $O(n^2)$ in very special cases, since it can simply mark all tiles of bombs. However, the average runtime for our algorithm is O(nlog(n)) since it is represented as a binary tree with at most a height of N. Since we do not need to fully expand the tree due to the pruning step, it is actually much faster than what we assumed. We will analyze and discuss the results of our algorithm in the latter sections.

Results:

In the graph below, it shows the relationship between time and number of threads. The x axis on the graph is the number threads being used in minesweeper solvers. The y axis on the graph is the average runtime of it being solved. The data was the average runtime of 1000 runs per thread. Each run is a 20x20 board with 100 bombs in the minesweeper. From the data below, it shows that serial time is a little slower than the parallel threads. However, with multiple threads, the difference seconds was not too drastic.

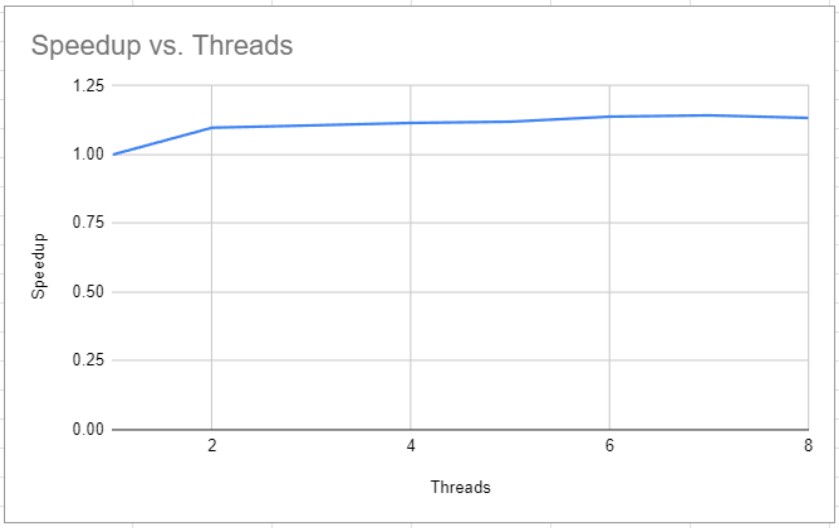
Thread count vs time	
Threads:	Time(Seconds)
1	2.76551
2	2.519
3	2.5
4	2.48
5	2.47
6	2.43
7	2.42
8	2.44

Testing was done using a 20x20
with 25% bomb count
400 tiles 100 bombs



In the graph below, it shows the relationship between speedup and the number of threads. The axis on the graph is the number threads being used in minesweeper solvers. The y axis on the graph is the average speedup of it being solved. The measurement of speedup is parallel thread runtime/ serial thread runtime. This graph shows a correlation that there is an increased speedup when adding more processes. This also shows that it is faster to parallel processes it, but adding more threads does not make too much of a difference.

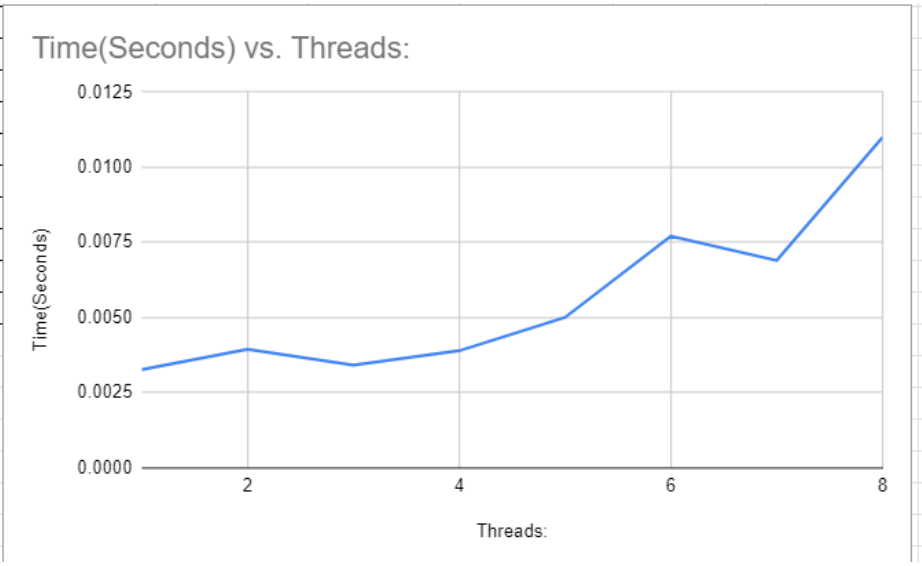
Speedup	
Threads	Speedup
1	1
2	1.097860262
3	1.106204
4	1.115125
5	1.119639676
6	1.138069959
7	1.142772727
8	1.133405738



In this graph we tested it using a 10x10 tile with 20% bomb count ran over 1000 trials and averaged their runtime performance.

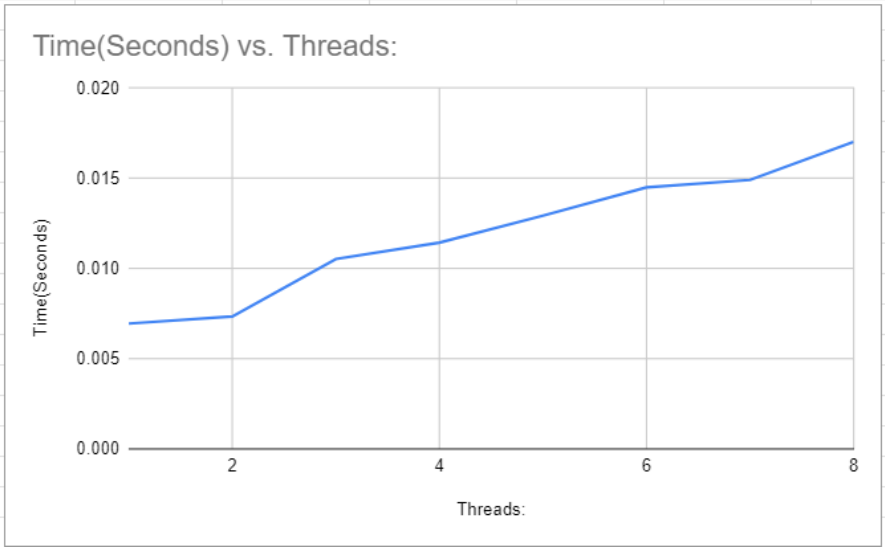
Thread count vs time	
Threads:	Time(Seconds)
1	0.00327
2	0.00395
3	0.00342
4	0.0039
5	0.00501
6	0.00771
7	0.0069
8	0.011

Testing was done using a 10x10
with 20% bomb count
100 tiles 20 bombs



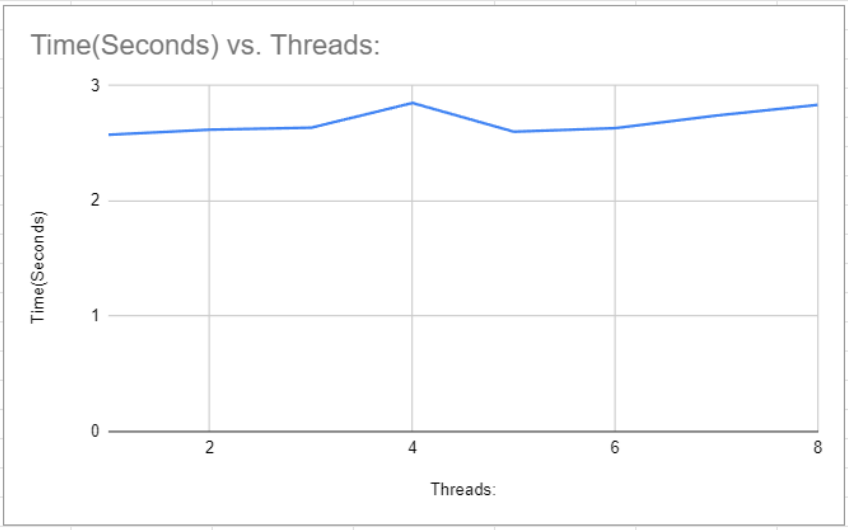
Thread count vs time	
Threads:	Time(Seconds)
1	0.00695
2	0.00734
3	0.01053
4	0.01144
5	0.01294
6	0.01451
7	0.01492
8	0.01704

Testing was done using a 15x15
with 20% bomb count



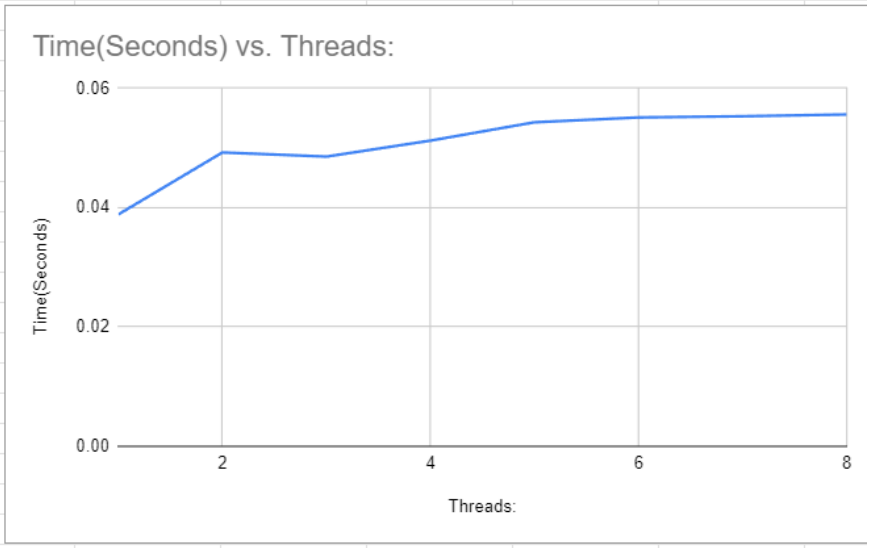
Thread count vs time	
Threads:	Time(Seconds)
1	2.575
2	2.6189
3	2.6362
4	2.8505
5	2.6014
6	2.6321
7	2.7416
8	2.8341

Testing was done using a 20x20
with 25% bomb count



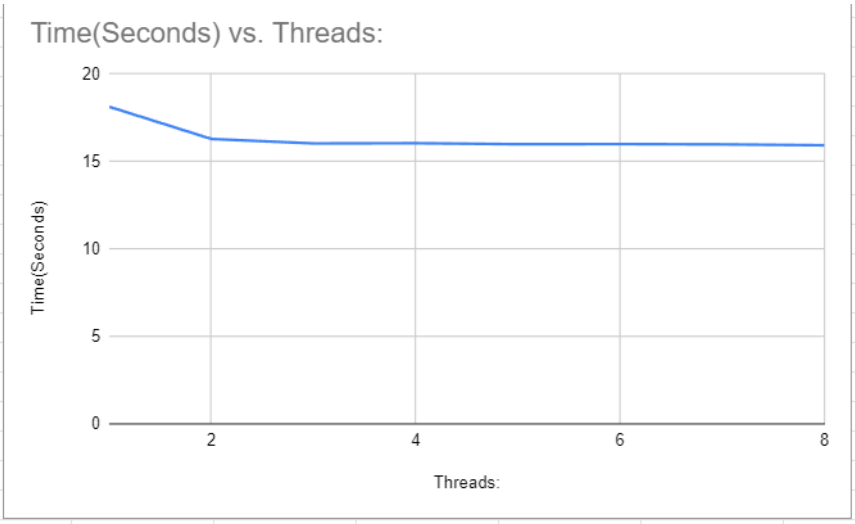
Thread count vs time	
Threads:	Time(Seconds)
1	0.03886
2	0.04925
3	0.04856
4	0.05124
5	0.05431
6	0.05512
7	0.05532
8	0.05562

Testing was done using a 25x25 with 20% bomb count



Thread count vs time	
Threads:	Time(Seconds)
1	18.133
2	16.2923
3	16.0369
4	16.0498
5	15.993
6	16.0014
7	15.983
8	15.9321

Testing was done using a 30x30 with 25% bomb count



Data Analysis:

Though there is a correlation between serial and parallel processing, there is little effect when adding more processes. This is most likely due to the matrix we evaluate in rule 3. Rule 3 overhead cost is a lot because when we check for probability, we would need to make multiple threads and data. Because of that, we have to create many deep copies of the board to avoid deadlocking and freeing our data. This in terms, results in a slower runtime. Because of this, it appears to consume more time and resources, and it shows that it is not significantly faster than the serial execution.

We also found through our testing that adding more bombs slows down the runtime. This is caused by the need to evaluate more bombs for rule 1, 2 and 3. Interestingly, when we increase the board size, the algorithm actually runs faster. To understand this, we need to understand the cost of each rule. In rule 1 and 2, the overhead cost is small because it only makes one new process to check if there are any neighboring bombs or safe tiles. It runs faster because the bombs are more likely spread out and the use of rule 1 and 2 is done really quick. Like we mentioned above, rule 3 has a lot of overhead cost, so avoiding it makes the runtime go a lot faster. With a bigger board, there is a lower probability of doing rule 3, which takes the longest. However, if we increase both the bomb and the board size, the runtime will be slower. This is because we will more likely evaluate rule 3 more than we did with either an increased board size or an increased number of bombs.

From doing this project, however, a minesweeper solver that is parallelized is unnecessary because evaluating rule 3 will take similar time to the serial version. A parallelized version of rule 1 and 2 does not make too much of a difference because the process is done very quickly. This results in a small difference in runtime and uses more processes for unnecessary reasons.

Judging by how the data suggested in the 10x10 grid says, threading actually costs us time due to the overhead of adding in more threads exceeding that of the cost of a smaller grid and edge cases. So a more optimal function would need to be calibrated such that if the grid size is small, then parallelization would not be applied but rather run in sequential time. We saw some parallelization performance improvements in the 40x40 grid, but not by much. So we can see parallelization improve performance when we see that the grid size gets larger and there are more unknown tiles to deal with.

Improvements:

One of the main issues we've encountered with our program is that we have made a mistake using C++ vectors, which are not thread-safe and may require a lucky run in order for it to actually fully process the data. This is due to the ADT of vectors having inconsistent removal and resizing. Since it is not threading-safe it might breach the buffer zone that a thread has allocated and get resized beyond what was allocated to a specific thread. As a result, we get a lot of segfaults and corrupted top size.

Upon closer inspection we noticed that there is significant overhead due to our algorithm performing a lot of checks before diving into its exploratory search. One way to improve the efficiency of this is to implement some type of memoization algorithm since if we implement rule 3, most likely the board state the next time we run it, will be similar to the same calculations we had before. This could drastically reduce the calculation and overhead of the algorithm.

In addition, we did not really implement a C++ GUI into this project. That way it is a bit more customizable, but we were focused on it being an abstract algorithm that can be implemented anywhere so the focus of this project was to create the algorithm instead of a fully functional app.

Appendices:

https://github.com/JWongCentral/Parallel_Minesweeper_Solver