Parallelizing a minesweeper solver with OpenMP

April 24, 2022

Jing Tao jt3149@nyu.edu
Sixing Zhou sz3704@nyu.edu
Yulin Hu yh2094@nyu.edu
Jingjing Bai jb7876@nyu.edu

Abstract

In this report, we presented an efficient parallel implementation to solve the minesweeper game. Our algorithm transforms the minesweeper game into Constraint Satisfaction Problems. By computing all the possible solutions of the constraints, the algorithm gets the best guess, the square with least probability of containing a mine to proceed. We tried two parallel implementation of the algorithm, one that decides the number of threads during runtime and one that has deterministic number of threads. Experimental results showed that both of the implementations performs well and gain huge speedups compared to the sequential version.

1 Introduction

Minesweeper is a popular game. The objective of the game is to clear a board with many unrevealed squares that might contain mines by reavealing non-mine squares and marking mines correctly. Revealed squares will provide information on its neighbors(how many mines are around this square) and you might be able to induce whether its neighbors contain a mine or not. The inference process of solving a minesweeper game can be transformed into solving a Constraint Satisfaction Problem naturally which is a classical decision problem that is studied widely. Iteratively solving the constraints, revealing new safe squares and getting new constraints, we will eventually solve the whole board. But success is not guaranteed even if you solved all the constraints correctly. There are times that there exists multiple possible solutions for the set of constraints and you have to make a guess to proceed.

In the following report, we implement a solver that maximizes the chance of winning by always solving for deterministic solutions first and when guessing is necessary, the solver makes the decision based on probability.

2 Background and Literature Survey

2.1 Solving minesweeper with CSP

A constraint satisfaction problem is defined as a $\langle X, D, C\rangle$, where $X = \{X_1, \ldots, X_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of their respective domains of values, and $C = \{C_1, \ldots, C_m\}$ is a set of constraints. In the minesweeper setting, the variables are the squares which we can denote by their Cartesian coordinates e.g. (0,1),(1,1).. The domain of every variable is equally $0, 1$ with 0 indicating not a mine, and 1 indicating a mine.

The number shown in a revealed square in a game indicates how many mines there are in the 8 adjacent neighbors which will be the right hand side of the constraint equations. Therefore, we can formulate a constraint with this given information. In figure1, We can see that square (0,3) shows 2. This gives us the constraint that $(0, 4) + (1, 4) = 2$. Similarly, the 3 in square $(1, 3)$ gives us the constraint $(0, 4) + (1, 4) + (2, 4) = 3$. These constraints are deterministic because there is only one solution possible when $(0, 4) = 1$, $(1, 4) = 1$, and $(2, 4) = 1$. This is called an "All mines neighbors" condition as all the neighbors are mines and we can safely mark them. Similarly, an "All free neighbors" condition is also deterministic since a 0 on the right hand side of the constraint equation just implies that there are no mines adjacent to this square and we can reveal them safely.

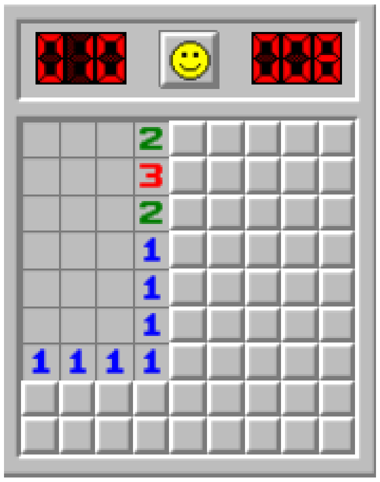

Figure 1: A 9*9 minesweeper game board with some squares revealed.

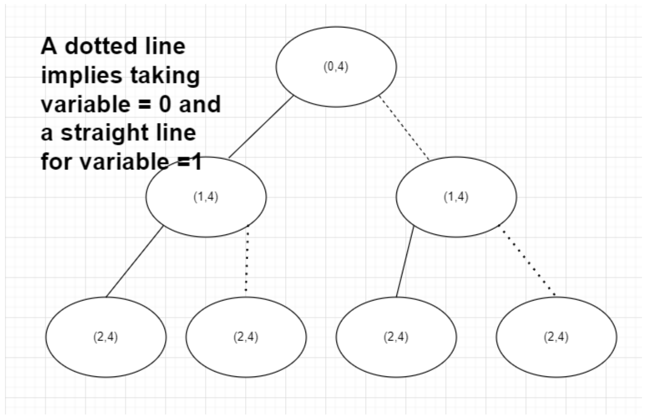
A dotted line implies taking variable = 0 and a straight line for variable =1
(0,4)
(1,4)
(1,4)
(2,4)
(2,4)
(2,4)
(2,4)

Figure 2: A binary tree generated by backtracking assignment

solution	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
x_{12}	0	0	0	1	1	1	1	1	1
x_{22}	1	1	1	0	0	0	0	0	0
x_{23}	0	1	1	0	1	1	0	1	1
x_{33}	1	0	1	1	0	1	1	0	1
x_{41}	0	0	0	0	0	0	1	1	1
x_{42}	0	0	0	1	1	1	0	0	0
x_{43}	1	1	0	1	1	0	1	1	0

Figure 3: All the solutions to the current constraints.

To solve a minesweeper game, we always proceed with those deterministic constraints which we call a deterministic step and reveal/mark the squares. When there is no deterministic steps available, we will solve all the constraints and get all the possible solutions. In a classic constrain satisfaction problem, people usually use recursive backtrack to find one viable solution. The backtrack algorithm will assign a value based on the domain of the variable and evaluate if this satisfies the constraints. If not it backtrack to the last assignment and try a different assignment. For a minesweeper game where every variable can only take two values, this will generate a binary tree like in figure 2. In the classic CSP, a backtrack algorithm finds one viable solution so it is going through a binary tree to find one leaf node. However, in the case of a minesweeper, if we only go with one potential solution, one problem is that the probability of this one being the real solution is slim as there might be huge numbers of solutions that can satisfy the current constraints. Another problem is that, given the solution for example $(1, 1) = 1$, $(1, 2) = 0$, $(2, 1) = 0$, we still cannot decide which square to reveal next because we don't know how "unsafe" each square is thus there is a big chance we will lose if we blindly random choose one square from the solution that is said to be non-mine to reveal. To overcome the above problems, we will compute all the viable solutions and based on the solutions, compute the square that has the lowest probability to be a mine to reveal next. In figure 3, we listed all the possible solutions at a specific time, given all the constraints we have. We will compute the probability of each variable being a mine and go with the lowest probability one to process to maximize the chance of winning. This is a indeterministic step and the runtime of a indeterministic step is $O(2^n - 1)$ where n is the size of variables from the constraints thus $2^n - 1$ nodes.

2.2 Related work

There are many studies on solving the minesweeper game with constraint satisfaction problem(CSP) techniques. Studholme in his unpublished report "Minesweeper as a Constraint Satisfaction Problem"([1]) discussed his CSP strategy for solving a minesweeper game but his solution doesn't perform well speed-wise because the original search space of the CSP problem encountered is too large. In his 600,000 games tested (on different board sizes), the largest problem solved had 69 variables, 30 constraints, and 1,620,880 solutions. The set with the largest number of variables and constraints had 100 variables, 67 constraints, and 9,240 solutions. The set with the largest number solutions was mentioned above and had 46 variables, 16 constraints, and 6,060,240 solutions. To reduce the search space, both Becerra and Raphael proposed to couple the individual variables to a bigger variable and solve the constraint with bigger variables to reduce the search space in their works([4], [5]). However, there is no mature way or algorithm to decide which variables to couple together so in the following report, we propose one way of coupling variables as an attempt to reduce the search space.

All the minesweeper solver works focus on the algorithmic side but not on the performance side. There isn't any parallel implementations on solving a minesweeper with CSP technique. On the other side, there are some attempts to speed up the CSP solver like in [2]. Vander-Swalmen and his crew propose to define a rich thread in charge of the search-tree evaluation and a set of poor threads that will help the rich one by simplifying the opened node. There implementation makes good memory allocation during the search. But those results don't really transfer to our study since we want all the solutions from the set of constraints and have to traverse the whole binary tree.

3 Proposed Idea

3.1 Algorithm

According to Studholme([1]), the problem size of the constraints grows sub-exponentially in the linear dimension. This can be understood intuitively because the variables that appear in the set of constraints to solve often appear to lie in a roughly straight vertical or horizontal line. In the settings of a classic expert difficult game, where the board is 30*16 with 99 mines. The problem size in the middle game would already be huge with 20-40 variables, 20-40 constraints which would take $O(2^{20})$ to $O(2^{40})$ time to traverse all the solutions.

Our attempt to reduce the problem size and make it viable is by computing all the possible solutions of the constraints at the given step and store them even if indeterministic step is not necessary. Therefore, when the step is a indeterministic step, we can build the binary tree based on the solutions we had from last step. Now we only assign and test values for the variables that are to the set of constraints after we reveal the square last step. At time step t, the number of viable solutions is s_t and the variable count is n_t. Naturally, we have $s_t < 2^{n_t}$, and in most real life cases $s_t << 2^{n_t}$ because only a small portion of the solutions satisfy the constraints. At time step t_{n+1}, originally, we would have to check $2^{n_{t+1}}$ solutions, but in our algorithm, we only check $s_t * 2^{n_{t+1}-n_t} = s_t * \frac{2^{n_{t+1}}}{2^{n_t}}$. So combine the two steps, the total computation is originally, $2^{n_{t+1}} + 2^{n_t}$ compared to our method $2^{n_t} + s_t \frac{2^{n_{t+1}}}{2^{n_t}}$. Since $s_t < 2^{n_t}$, our method is guaranteed to less computations in those two steps. By induction, we can extend this result to the every iterations.

The pseudo code is presented below in algorithm 1.

3.2 Dynamic thread scheduling method I

For the csp solver, the number of main loop is unknown before code's actual running. Judgement of stepping on mines is set to control the main loop's terminal. So it is necessary to divide the main loop into subproblems and parallelize the for loop individually.

The number of threads created and destroyed for each for loop is also count, especially when traversing the neighboring grid around a certain unknown one on the boundary. The number of loop vary a lot for different problem size, grid location or even random mines decided by different seed. So using alterable parameter for various loop is will better fit the problem.

A simple algorithm for this is to schedule thread and cpu cores as few as possible. We set a minimum iteration number for each thread (for this problem this value is 5), so the number of threads use is ceiling(loop nums / minimum iteration). Also a maximum thread num is also set as double cpu core nums. This method largely boost efficiency because it can better load balance. When thread nums far exceeds the cpu cores num, frequent task switching and scheduling exhausted cpu performance. Similarly, too few threads lead to sequential calculation.

Algorithm 1 Solve minesweeper by CSP

```
constraints ← ()
variables ← ()
newVariables ← ()
solution ← ()
solutions ← ()
previousSolutions ← ()
while Game is not finished or failed do
  Update constraints
  Update variables
  Check for deterministic step
  if we have a deterministic solution then
    add deterministic squares to solution
    guess = false
    add deterministic squares to previousSolutions
  end if
  newVariables = variables - variables in previousSolutions
  k=size of newVariables
  for index from 0 to k do
    if assignment satisfy the constraint then
      add assignment to solutions
    end if
  end for
  if guess=true then
    compute the square with least probability being a mine from solutions
    add that square to the solution
  end if
  Mark and reveal the squares we have in solution
  previousSolutions = solutions
end while
```

3.3 Dynamic thread scheduling method II

For the sequential version, there is a main loop that has 2^n iterations and it will be huge when n gets bigger. Besides, each iteration needs lots of computations when the number of constraint equations gets larger. So the main problem is to reduce the execution time for this loop. Here we used for directive to let it run parallel. Since the computations in each loop vary and are not linearly increasing or decreasing, we used schedule clause with type of dynamic.

Also since other loops have situations that different threads could access the same variable or the same location of memory, we need to make use of critical directive or atomic directive. After some testing, we find the overheads of the directives are not worth to parallelize the loops that don't have much computations, so we decide to give up parallelizing these loops.

Different from previous method, we omitted the computations for the number of threads needed and just use the number from command line argument as the number of threads for this whole program.

4 Experimental Setup

We conducted experiments on a Linux server with 4 AMD Opteron 6272 16-core CPU@2.1GHz running CentOS 7 operating system. The per core L1 and L2 cache sizes for this CPU are 768KB and 16MB respectively, and the shared L3 cache size is 16MB. The memory size is 256 GB. All code is written in C++ and compiled using G++ 4.8.5 with C++ version flag -std=c++11 flag.

5 Experiments & Analysis

We have coded two versions of parallel implementation.

5.1 Results

- Method I

  For method I, dynamic thread schedule is used and running time contrast is listed below in Figure 4 to 6. x-axis is assigned with problem size and different boards generated. We can see that the actual running time show a exponential order to the linear problem size increase.(For figure 5, the parallel version finishes in 0.02 seconds, making the graph very hard to see).

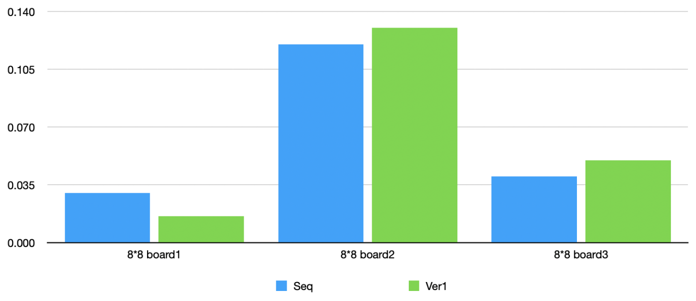

Figure 4: Board size 8* 8 Contrast between sequential and parallelized version

- Method II

  For method II, We tested sequential code and parallel code with different sizes (same seed) and different numbers of threads. Since with specific seed, the game will always produce same result: either success or fail, we only pick the certain seed that will produce success result. We also used time command to get the real part as total execution time and to make comparison between sequential code and parallel code, we took speedup as performance metric. To get more precise numbers and avoid fluctuations, we repeat each measurement 3-5 times and take the average as the total execution time.

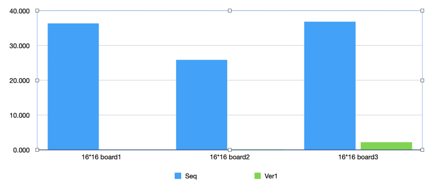

Figure 5: Board size 16*16 Contrast between sequential and parallelized version

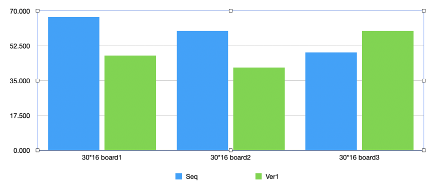

Figure 6: Board size 30*16 Contrast between sequential and parallelized version

We have three levels of difficulty for minesweeper game: beginner(8*8 with 12 mines), intermediate(16*16 with 51 mines) and expert(30*16 with 96 mines). With certain seed mentioned above, we have information for the total execution time of sequential code.

		Sequential Time(s.)		
Width	Height	Min	Max	Mean
8	8	0.063	0.064	0.0637
16	16	19.643	19.771	19.650
30	16	34.250	34.809	34.462

Figure 7: sequential runtime

Following this logic, we obtained the total execution time with different numbers of threads for each level of difficulty and used the formula $Speedup = T_s \ / \ T_p$ to get corresponding speedup.

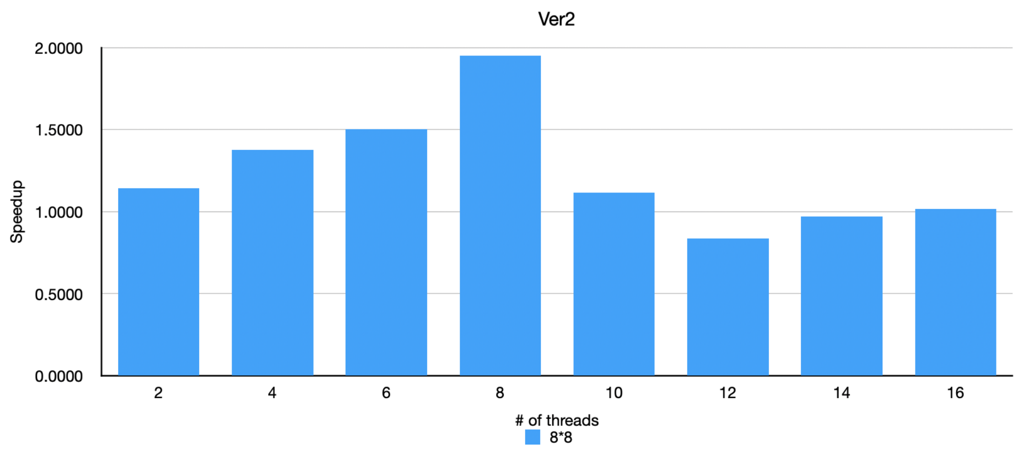

Figure 8: 8*8 board size

5.2 Analysis

- Method I

Although the actual running time has large fluctuations due to various seeds, it is obvious that parallelized version gain great superior to the sequential one especially for the 16*16 problem size. For the 8*8 and 30*16 version, the problem size is far too small or far too large, parallelization can't gain splendid performance because of the overload or vacancy of cpu cores.

It also can be found that the number of scheduling threads change with the number of the vars.size(), which is a parameter related to the real-time board status and current number of loops. This can be checked when running the code and threads number will be printed to the terminal with '#' ending. This method uses extra expenses but the superior performance

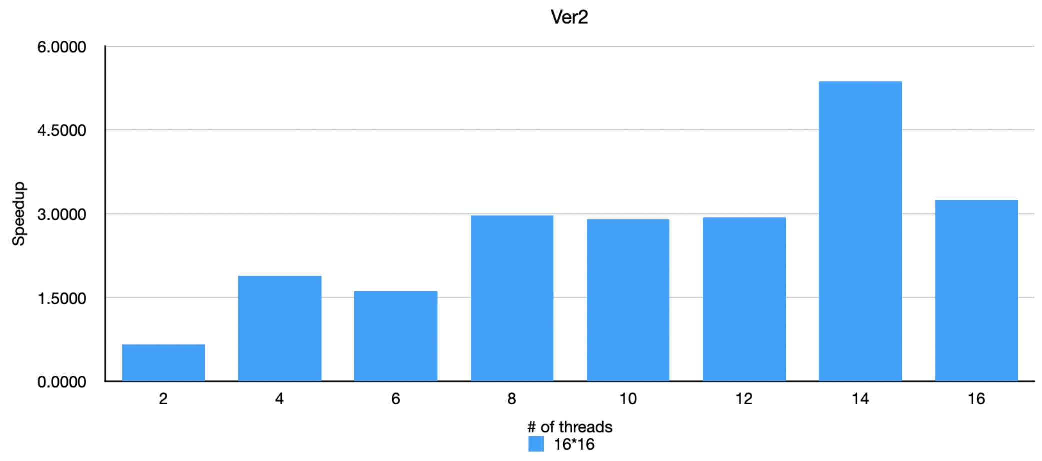

Figure 9: 16*16 board size

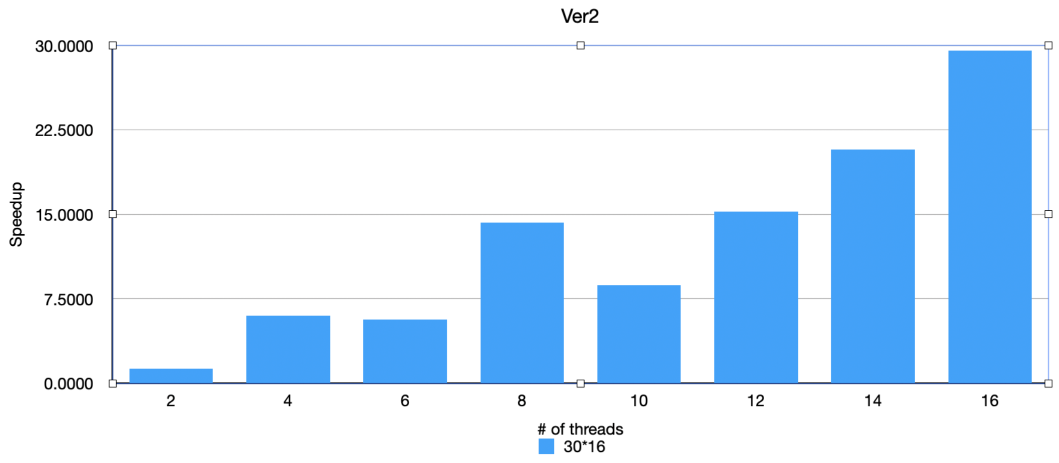

Figure 10: 30*16 board size

when problem size is too large or too little deserve it. The code will gain more robustness and readability in this way.

- Method II

  For 8*8 board size (figure 8), we found the speedup reached the maximum when the number of thread = 8; For 16*16 board size (figure 9), we found the speedup reached the maximum when the number of thread = 14; For 30*16 board size (figure 10), we found the speedup is the maximum when the number of thread = 16. So as we know, when board size gets bigger, some certain iterations will have more constraint equations, which makes loops increase in exponential way. So when the number of threads could make the schedule clause assign balanced workload for each thread and thus reduce the wait time for completions of all iterations, the speedup will reach the highest and when it passes this point, the speedup drops since workload becomes imbalanced again. For 8*8 board size and 16*16 board size, we could find this trend. But for 30*16 board size, we find when the number of threads = 2^n, the speedup will get higher compared to the number of threads = $2^n \pm 1$. And after some testing, we discovered the speedup could be higher if the number of threads is more than 16. As mentioned above, the number of iterations increases in exponential manner, so when the number of threads gets close to 2^k, the workload balance would be better and thus the speedup is comparatively good.

6 Conclusions

1. Comparing the two parallel methods, the one that uses deterministic thread count seems more robust. Hard-coded may have better performance than flexible ways confronted with certain problems.

2. Combination of software and hardware is more important than programming extensive when applied on certain machine. Hardware is progressing with each passing day and we shall pay more attention on hardware capabilities other than the programming itself.

References

[1] Chris Studholme *Playing the Minesweeper with Constraints*, unpublished.

[2] Pascal Vander-Swalmen, Gilles Dequen, and Micha¨el Krajecki (2008) *On Multi-threaded Satisfiability Solving with OpenMP*, OpenMP in a New Era of Parallelism.

[3] Allan Scott, Ulrike Stege, Iris van Rooij (2011) *Minesweeper May Not Be NP-Complete but Is Hard Nonetheless*, Math Intelligencer 33, 5–17 (2011). https://doi.org/10.1007/s00283-011-9256-x.

[4] Becerra, David J, *Algorithmic Approaches to Playing Minesweeper*, unpublished,

[5] Raphaël Collet (2004) *Playing the minesweeper with constraints*, In Proceedings of the Second international conference on Multiparadigm Programming in Mozart/Oz (MOZ'04). Springer-Verlag, Berlin, Heidelberg, 251–262. $DOI : https : //doi.org/10.1007/978-3-540-31845-3\_21$