Parallelization of DPLL Algorithm to Solve the N-Queen Problem

Ali Yasser Ismail

April 27, 2014

1 Introduction

Formally, SAT solvers are algorithms that determine if there exists an interpretation that satisfies a given boolean formula[1]. Informally, SAT solvers can reason through problems that are defined through boolean formulas. An example, is the well known the n-Queen problem which asks "Can and where must n queens be placed on a chess board so they do not attack each other [2]?" This problem can be defined with a boolean formula and then solved with a SAT solver.

Of course, there are more practical uses for SAT solvers. They can be used for hardware/software verification, automated reasoning, and also as search engines [3]. The question we raise is, can a SAT solver gain even greater performance by being parallelized? More specifically, can the SAT solving algorithm DPLL be parallelized, thus leading to greater performance in solving the n-Queen problem?

2 Motivation

The biggest motivation for this project is to measure the impact of concurrent programming on the performance of a basic SAT solving algorithm. There are many SAT solving algorithms with many optimizations. Even after all the optimizations have been made, can the performance reach greater depths with parallelization? If the answer is yes, then it would be worth while to parallelize SAT solvers to achieve performance that currently seem out of reach.

3 Objective

The first objective was to parallelize DPLL for n threads. Next, the performance was measured for 1, 2, 3, and 4 threads. The n-Queen problem served as the payload and execution time as the performance measure.

Each quantity of threads was given 4 to 10 queens to solve and was timed for each payload. This gave an insight on the effects of introducing multiple threads to solve a boolean formula, using DPLL. Also, this experiment reflected how well the DPLL algorithm was parallelized. In addition, the final objective was to optimize a sequential version of the DPLL algorithm and compare it to the concurrent version.

4 Original Approach

4.1 Problem

As mention above, the first objective was to parallelize the DPLL algorithm below. In general, DPLL is a backtracking algorithm that inserts TRUE or FALSE for each variable in a boolean formula and recursively checks if the formula can be satisfied.

```
1 let rec DPLL F =
2   let F' = BCP F in
3   if F' = ⊤ then true
4   else if F' = ⊥ then false
5   else
6     let P = CHOOSE vars(F') in
7     (DPLL F'{P ⟼ ⊤}) ∨ (DPLL F'{P ⟼ ⊥})
```

Lines 1 to 6 are easily parallelized as long as each of the threads have their own formula to reduce and manipulate. The trouble occurs with line 7[4]. The basic DPLL algorithm does not consider multiple threads sharing the workload to determine satisfiability for a boolean formula.

Sequentially, a thread passes TRUE and, if needed, FALSE for a variable to the DPLL algorithm. The DPLL algorithm then takes this value, reduces the boolean formula, checks if the formula has been solved and, if it has not, it calls itself again with TRUE and, if needed, FALSE. Eventually, the algorithm will reach a point where the guesses satisfy or do not satisfy the formula. If the formula is not satisfied, the algorithm will back track and continue guessing until it finds the correct sequence of guesses.

Figure 1 shows this process graphically. The algorithm begins at x1. First it guesses TRUE, reduces the boolean formula, checks satisfiability, and, if satisfiability has not been concluded, it moves on to x2. Now the algorithm guesses TRUE again, reduces the formula, but realizes that this does not satisfy the formula. The algorithm then back tracks to x2 where it was valid and can make another guess, this time it will guess FALSE. If this guess does not satisfy the formula then the algorithm will back track to x1, otherwise, it goes deeper into the tree in search of a satisfiable sequence of guesses.

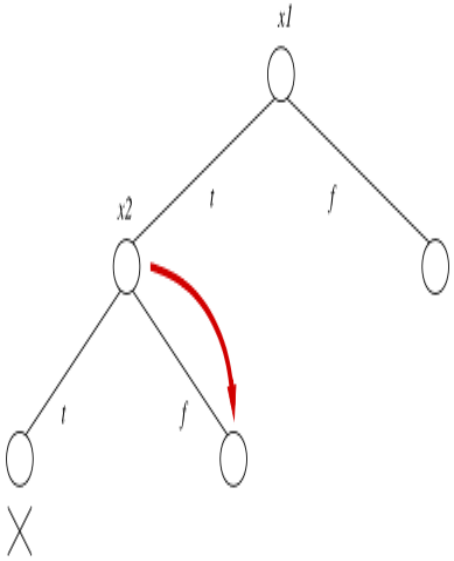

Figure 1: DPLL recursive steps [5].

4.2 Single Concurrent Queue Solution

The question is then, how can line 7 be manipulated to allow multiple threads to share the workload to achieve the ultimate goal of solving the n-Queen problem? Instead of a thread returning from DPLL F'(P $\longmapsto \top$) and moving on to solve DPLL F'(P $\longmapsto \bot$), the thread is allowed to queue the work to solve DPLL F'(P $\longmapsto \bot$) and move on to verifying DPLL F'(P $\longmapsto \top$). Other threads can then dequeue the work and help complete searching for a satisfiable sequence.

Figure 2 shows the process. Thread T1 queues the guess FALSE for the first variable. Then it continues to solve satisfiability for the guess TRUE for that first variable. The guess does not solve the question of satisfiability so thread T1 moves on to the next variable. It queues the guess FALSE for the next variable and moves on to solve satisfiability for the variable next in line. While T1 solves its own part of the tree, thread T2 dequeues the sequence of variables queued by T1 and begins solving the tree and queuing guesses along the way.

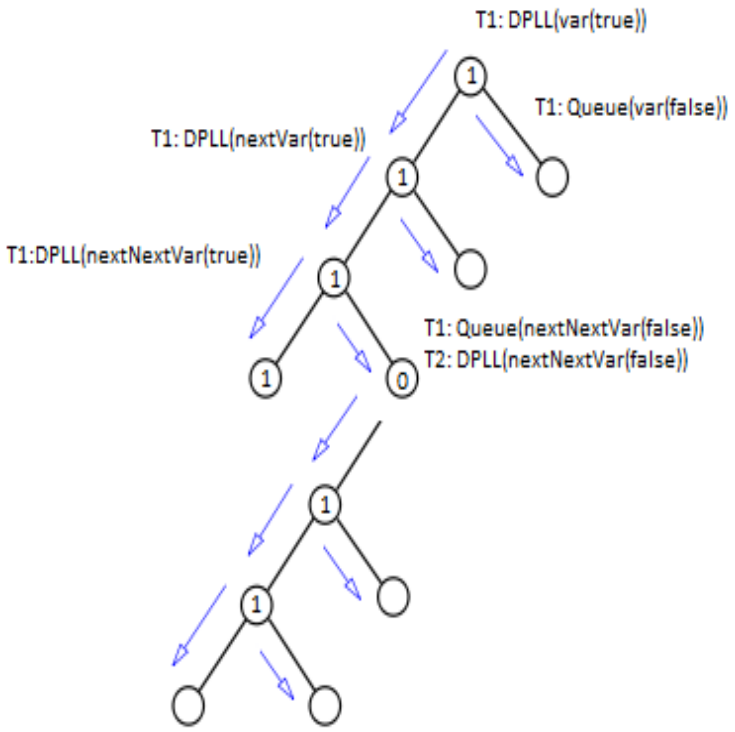

Figure 2: Modified DPLL recursive steps [5].

More specifically, take a look at the modified DPLL algorithm below. The algorithm begins with a kick start section. There must only be one thread that starts the DPLL algorithm so the others can extract work from the TASKS queue, which is a concurrent queue provided by the ConcurrentLinkedQueue class[6]. The thread chooses the first variable from a list of variables, stores FALSE for this variable in the TASKS queue, stores TRUE for the variable in a buffer called STATE and continues to the next variable. In the next iteration, this thread will skip the kick-start section, append FALSE to its current state, queue its state in the TASKS queue, and move on to solve for the current variable. STATE is a buffer that tracks the previous guesses for a thread. This way, when a different thread pulls from the TASKS queue, they will have a list of variables that were guessed up to that point and they can continue the search.

For example, take a look at Figure 2. When thread T2 dequeues from the TASKS queue, it cannot simply guess TRUE for the nextNextVar. It needs to evaluate the boolean formula on the previous guesses TRUE, TRUE, TRUE, and FALSE.

```
let rec DPLL F =
  let F' = BCP F in
  if F' = ⊤ then true
  else if F' = ⊥ then false
  else
    let P = CHOOSE vars(F')
    if STATE is EMPTY
      STORE P ⟼ ⊥ in TASKS
    else
      STORE P ⟼ ⊥ in STATE
      STORE STATE in TASKS
      REMOVE P ⟼ ⊥ from STATE
    P ⟼ ⊤
    STORE P in STATE
    DPLL {F'{P ⟼ ⊤}, STATE}
```

In addition, work extraction and state evaluation is done before a thread calls DPLL. The algorithm below is the outer shell that is run by each thread. Each thread begins by waiting for Thread 0 to generate the formula for the n-Queen problem, then generate a list of all the variables, and place the first task in the TASKS queue. When Thread 0 does all of this, the rest of the threads will enter and attempt to grab a task from the queue. When they are successful, they will evaluate the state of their task. If the state of the task is satisfiable and consists of guesses for the entire formula(return of 1) it then sets the "answer" flag to TRUE to let other threads know a solution has been found. If the state is not satisfiable(return of -1), it then continue on to grab another task from the queue. If satisfiability of the formula is not solved (return of 0), it will continue on to solve for satisfiability by calling BCP and DPLL.

```
let  TestDPLL =
    while THREAD_ID != 0 && !(THREAD 0 STORE P ⟼ ⊥ in TASK )
    if THREAD_ID == 0
      createFormula()
      createVarList()
      BCP(formula, varList)
      if DPLL(formula, varList, TASKS, past) == TRUE
        answer = TRUE
    while !answer
      COPY formula
      COPY varList
      DEQUEUE STATE from TASKS
      if evalState(formula, varList, STATE) == 0
        BCP(formula, varList)
        if DPLL(formula, varList, TASKS, past) == TRUE
          answer = TRUE
      else if evalState(formula, varList, STATE) == 1
        answer = TRUE
```

In summary, the goal of this algorithm is to allow n threads search different parts of the guess

tree, as shown in Figure 2, to determine a state where the boolean formula is satisfiable, thus giving a solution to the n-Queen problem.

4.3 Multiple Concurrent Queue Solution

The single concurrent queue solution allows the n-Queen problem to be split up between n threads, however, it can be further optimized to increase the productivity of each thread. In order to do so, a concurrent queue is allocated for each thread. If a threads TASKS queue is empty, they traverse the queues of the other threads to find work. When they find work, they will fill their own TASKS queue with tasks that they will work on later. This reduces collisions between threads when more than one thread must access a queue, forcing the others to attempt to dequeue from the queue again.

Accommodating queues for each thread does not take much alteration to what was described above. Below is the modified TestDPLL algorithm that each thread runs to solve the n-Queen problem.

```
1  let  TestDPLL =
2      while THREAD_ID != 0 && !(THREAD 0 STORE P ⟼ ⊥ in TASK )
3      if THREAD_ID == 0
4        createFormula()
5        createVarList()
6        BCP(formula, varList)
7        if DPLL(formula, varList, TASKS.get(ID), past) == TRUE
8          answer = TRUE
9      while !answer
10       COPY formula
11       COPY varList
12       if TASKS.get(ID) is !EMPTY
13         DEQUEUE STATE from TASKS.get(ID)
14       else
15         while TASKS.get(ID) is EMPTY
16           for k = 0, k < n, k++
17             DEQUEUE STATE from TASKS.get(k)
18       if evalState(formula, varList, STATE) == 0
19         BCP(formula, varList)
20         if DPLL(formula, varList, TASKS.get(ID), past) == TRUE
21           answer = TRUE
22       if evalState(formula, varList, STATE) == 1
23         answer = TRUE
```

The only modification is that now TASKS is a list of concurrent queues, one for each thread. Threads now pull tasks from their own queue, unless they are empty. If they are empty, they traverse the TASKS list and attempt to dequeue from the queues of other threads. Also note that the TestDPLL and DPLL algorithms are written specifically to solve the n-Queen problem. The assumption is that a solution will eventually be found, therefore, the algorithm will eventually terminate. What is not shown in the pseudocode above is the series of checks

of the answer flag that indicates when a solution is found. When that is set to TRUE, the other threads will eventually check that flag and terminate.

5 Original Approach Experiments

5.1 Experiment 1

The first experiment is based on the concurrent DPLL algorithm and runs as follows:

- Test 4 to 10 Queens Problem on 1, 2, 3, and 4 threads (Time Executions)

This experiment reflects how concurrent the DPLL algorithm is and how well it performs in solving the n-Queen problem. If DPLL is mostly parallelized, then execution time should decrease as the number of threads solving the problem increases.

5.2 Experiment 2

The second experiment is based on the unmodified DPLL algorithm and runs as follows:

- Test 4 to 10 Queens Problem (Time Executions)

This purpose of this experiment is to see how much worse or better the sequential version of DPLL executes the same payload as the concurrent DPLL.

6 Results

N-Queens/Threads	1	2	3	4
4	170 ms	140 ms	145 ms	160 ms
5	500 ms	370 ms	400 ms	520 ms
6	1450 ms	700 ms	690 ms	845 ms
7	9460 ms	2660 ms	2635 ms	3205 ms
8	79039 ms	10400 ms	13400 ms	15125 ms
9	260925 ms	36710 ms	48885 ms	130915 ms
10	1900595 ms	258645 ms	357685 ms	460755 ms

Table 1: Execution times for the concurrent queue DPLL algorithm (1^{st} Experiment).

N-Queens/Threads	1
4	105 ms
5	155 ms
6	260 ms
7	290 ms
8	735 ms
9	1080 ms
10	2050 ms

Table 2: Executions times for the sequential DPLL algorithm (2^{nd} Experiment).

7 Original Analysis

Based on experiments 1 and 2, it seems that my concurrent version of DPLL does not even compare to the sequential version. The sequential version is much faster, however, there is still hope. It is very possible that the threads of the concurrent version spend too much time queuing and dequeuing from their task queues.

Notice in Table 1 that the 2 threads running the concurrent DPLL is the fastest in almost all of the categories, with 3 and 4 threads falling slightly behind. The latency caused by the queue accesses could be the major factor in this result. The sequential version does not waste time accessing any data structure and goes hard at work to solve for satisfiability. Based on this, it may be advantageous for the concurrent version to spend more time doing work like the sequential version. We propose that the concurrent version has the potential to be faster than the sequential version.

If the concurrent version spent less time queuing and dequeuing and spent more time solving, it could get an answer faster than the unmodified DPLL. Why is this? As shown in Figure 2, the DPLL algorithm is traversing a tree of guesses trying to find the correct sequence that solves satisfiability. Depending on the size of the n-Queen problem, the tree can become very large, therefore, it intuitively seems guaranteed that if the tree was split only into two tasks, two algorithms can go hard at work solving the algorithm. In this scenario, there would be no need for a queue. One thread would solve DPLL (F'(P $\longmapsto \top$)) and the other would solve DPLL (F'(P $\longmapsto \bot$)). This should reduce execution time, however, this only takes advantage of two threads. Can this be extended to n threads more efficiently than my first attempt?

8 New Approaches

As discussed earlier, the first attempt at parallelizing DPLL was not a successful one, however, the problem was identified. There are too many accesses to queues by the threads, causing major delay in solving the n-Queen problem. Now the question is, can the queues

be eliminated and/or can this new parallelized algorithm be extended to handle n threads?

8.1 No Concurrent Queue Approach

As mentioned in Section 7, a suggestion was made to allow one thread to solve DPLL (F'(P $\longmapsto \top$)) and the other would solve DPLL (F'(P $\longmapsto \bot$)). This is a very simple approach and does not require any modification to DPLL. The threads just make their respective calls to DPLL and the time it takes for them to find a solution is recorded. Note that it is not necessary to record the execution time of searching the entire search space. It is only necessary to record the time it takes for a solution to be found.

8.2 Modified Concurrent Queue Approach

In order for n threads to share work, there must be a data structure they access to extract work. With this realization, concurrent queues must be used , however, this does not mean queue accesses cannot be reduced. In order to reduce the number of times a thread must access their queue, the original DPLL algorithm is used, rather, than the concurrent DPLL algorithm. Instead of a thread adding to the task queue after every guess, it adds only after the first guess. This will decrease the amount of work that goes into the queues and keep the threads working away at solving satisfiability.

Figure 3 shows the new method graphically. Thread 1 begins by adding FALSE for the first variable into its task queue and goes to work solving the whole left side of the tree. Before, thread 1 only solved a single path and had to access its task queue to solve another path. Now threads are solving whole branches of the tree. This reduces the amount of access to the task queues and increases the time spent towards solving satisfiability. As mentioned earlier, the original DPLL algorithm is used, therefore, modification was done only to Test-DPLL, which can be observed below.

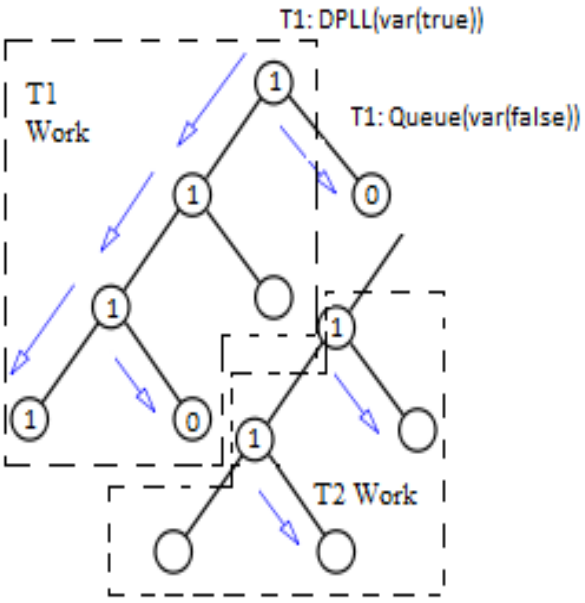

Figure 3: New concurrent approach to solve satisfiability.

```
let  TestDPLL =
    if THREAD_ID == 0
      createFormula()
      createVarList()
      BCP(formula, varList)
      if DPLL(formula, varList, TASKS.get(ID), past) == TRUE
        answer = TRUE
    else
      while THREAD_ID != 0 && !(THREAD 0 STORE P ⟼ ⊥ in TASK )
      while !answer
        COPY formula
        COPY varList
        if TASKS.get(ID) is !EMPTY
          DEQUEUE STATE from TASKS.get(ID)
        else
          while TASKS.get(ID) is EMPTY
            for k = 0, k < n, k++
              DEQUEUE STATE from TASKS.get(k)
        STORE P ⟼ ⊥ in STATE
        STORE STATE in TASKS.get(ID)
        REMOVE P ⟼ ⊥ from STATE
        if evalState(formula, varList, STATE) == 0
          BCP(formula, varList)
          if DPLL(formula, varList, TASKS.get(ID), past) == TRUE
            answer = TRUE
        if evalState(formula, varList, STATE) == 1
          answer = TRUE
```

9 New Approach Experiments

Unlike previous experiments, these experiments were run on the ecen5033.colorado server, an 8-processor machine. Doing so maintains consistency and allows for increasing the thread count up to 8 threads.

9.1 Experiment 3

The third experiment is based on the no concurrent queue approach and runs as follows:

- Test 4 to 14 Queens Problem on 2 threads (Time Executions)

This experiment will show how much speedup is achieved when two threads are running the original DPLL algorithm without ever having to access a data structure. The ultimate goal is to achieve a lower execution time for the same workload as the original sequential DPLL algorithm. This alone will show the benefits of effective parallel programming. Also, extending this experiment to 14 queens will be discussed in the next section.

9.2 Experiment 4

The fourth experiment is based on the modified concurrent queue approach and runs as follows:

- Test 4 to 14 Queens Problem on 2, 3, 4, and 5 threads (Time Executions)

First note that one thread was not tested since the purpose of this approach is to reduce execution time by introducing multiple threads. Also, the goal of this experiment is to compare its execution time to that of the no concurrent queue approach. As mentioned earlier, these experiments were extended to 14 Queens. The reason for this is to test the possibility that the modified concurrent queue algorithm performs better when n-Queen workload is large.

9.3 Experiment 5

The fifth experiment is identical to experiment 2, however, it was extended to solve 14 queens.

- Test 4 to 14 Queens Problem (Time Executions)

Since experiments 3 and 4 were ran on a different machine than 1 and 2, experiment 2 was rerun to obtain a true and consistent comparison to the performance of the sequential DPLL algorithm.

10 New Results

N-Queens/Threads	2
4	54 ms
5	83 ms
6	126 ms
7	182 ms
8	330 ms
9	1095 ms
10	2755 ms
11	5098 ms
12	12051 ms
13	20336 ms
14	65168 ms

Table 3: Execution times for the no queue concurrent DPLL algorithm($3rd$ Experiment).

N-Queens/Threads	2
4	0.96
5	0.83
6	1.08
7	0.85
8	1.34
9	0.60
10	0.52
11	0.92
12	0.89
13	0.87
14	0.85

Table 4: Speed-up for the no queue concurrent DPLL algorithm($3rd$ Experiment).

N-Queens/Threads	2	3	4	5
4	57 ms	57 ms	57 ms	51 ms
5	82 ms	73 ms	67 ms	61 ms
6	104 ms	117 ms	107 ms	120 ms
7	194 ms	179 ms	220 ms	251 ms
8	471 ms	486 ms	483 ms	494 ms
9	1042 ms	874 ms	935 ms	1094 ms
10	2527 ms	2385 ms	2246 ms	2268 ms
11	4688 ms	4175 ms	4272 ms	5009 ms
12	10356 ms	9527 ms	11411 ms	9779 ms
13	10471 ms	13762 ms	17539 ms	22455 ms
14	56014 ms	38346 ms	52686 ms	42641 ms

Table 5: Execution times for the modified concurrent queue DPLL algorithm ($4th$ Experiment).

N-Queens/Threads	2	3	4	5
4	0.91	0.91	0.91	1.01
5	0.84	0.95	1.02	1.13
6	1.30	1.16	1.27	1.13
7	0.80	0.87	0.70	0.61
8	0.94	0.90	0.91	0.89
9	0.63	0.75	0.70	0.60
10	0.56	0.60	0.63	0.63
11	0.99	1.11	1.09	0.93
12	1.03	1.12	0.93	1.09
13	1.68	1.28	1.00	0.78
14	0.99	1.45	1.05	1.30

Table 6: Speed-up for the modified concurrent queue DPLL algorithm ($4th$ Experiment).

N-Queens/Threads	1
4	52 ms
5	69 ms
6	136 ms
7	155 ms
8	441 ms
9	655 ms
10	1420 ms
11	4664 ms
12	10689 ms
13	17610 ms
14	55543 ms

Table 7: Execution times for the sequential DPLL algorithm ($5th$ Experiment).

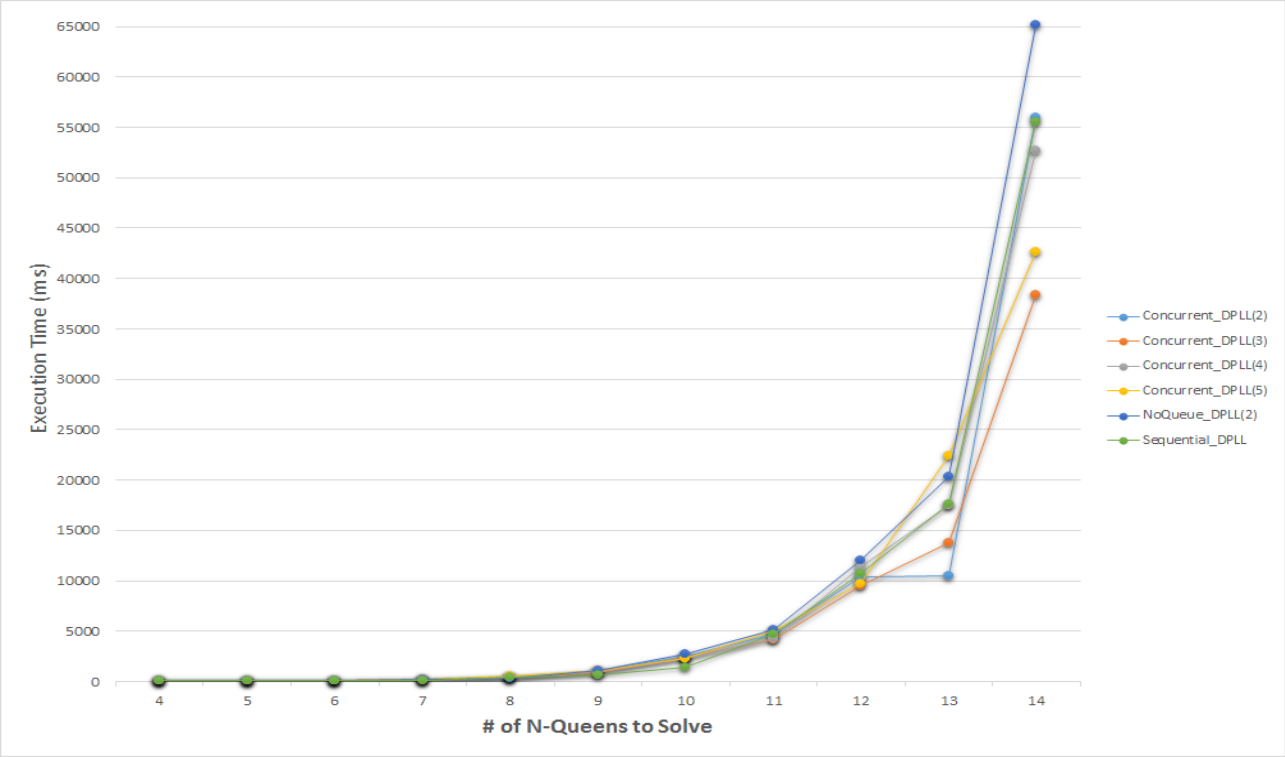

Figure 4: Graph of all experiment results.

11 New Approach Analysis

Based on the overall results shown n Figure 4, my hypothesis was correct. By reducing the amount of accesses to a queue, the execution time of the concurrent DPLL algorithm was reduced. That was a success, however, we have not achieved my ultimate goal of gaining significant speed-up. Tables 4 and 6 show the speed-ups for every scenario presented to

the current queue and no concurrent queue DPLL algorithms. There are many cases where speed-up exceeds 1, however, nothing goes above 2 even though there are scenarios where more than 2 threads are used. There are also many cases where speed-up dropped below 1. In addition, Figure 4 gives a graphical view of how each algorithm performed with respect to each other. There is not a significant different between execution times, even when 5 threads are used. The question now is, what could have went wrong?

11.1 Solution Location

The objective of a SAT solver is determine satisfiability for a boolean formula. When a SAT solver achieves this, it quits. Consider the discussion on how DPLL works in Section 4. Figuratively speaking, it generates a tree of guesses for all variables. For the sake of simplicity, imagine this tree already exists and DPLL simply searches the tree until it finds the right sequence for the n-Queen problem. There may be multiple answers in the tree, however, DPLL will quit when it finds the first one.

Now consider the new approach concurrent DPLL algorithm as described in Section 8. The tree is essentially fed to multiple threads that help locate a solution to the n-Queen problem. In fact, the left side of the tree is left entirely for a single thread to solve, which is shown in Figure 3. Remember, the reason for this was to minimize the amount of queue accesses. Unfortunately, in an attempt to minimize queue accesses, a new problem arose. If the concurrent DPLL deploys only a single thread to the left side of the tree, then, with respect to the left side of the tree, there will not be a difference between the sequential and concurrent DPLL algorithm. Remember that the sequential algorithm starts its search on the left side of the tree. One could still argue this phenomenon is okay because it will still take the sequential algorithm time to search the right half of the tree. This would be true if the solution to the problem was only on the right side of the tree. If the satisfiability can be determined just by looking at the left side of the tree then no speed-up will occur. In fact, the concurrent DPLL will take longer because it needs time to setup an infrastructure so that the threads can communicate and work steal.

All in all, the location of the solution in the tree plays a significant role in how well the concurrent DPLL will do. If an instance of an answer occurs early on the right side of the tree or if the first instance of an answer occurs on the right side of the tree, the concurrent DPLL algorithm will perform better. Note that there are multiple answers, depending on the number of queens being solved for.

11.2 Overhead

Similarly, by introducing an n amount of threads, there is a lot of work to be done to ensure they do not do the same work and/or they get work when they need it. To solve these issues, we used a list of concurrent queues. This allowed each thread to maintain their own workload and traverse the queues of other threads to extract work if they had nothing do. While this spreads works, it creates overhead. While the multiple threads running the concurrent DPLL are traversing the queues, the single thread running the sequential DPLL is hard at work.

This could have played a powerful role in the small amount or no speed-up achieved by the concurrent DPLL.

11.3 Conclusion

All in all, this project was a partial success. We was able to parallelize the DPLL algorithm using a concurrent queue as a foundation. Multiple threads pushed work on their queues and then they went on to solve satisfiability on a small amount of work. If they ran out, they would grab more work from their queues or the queues of other threads. While this approach seemed effective, it proved to be a slight failure, as can be observed in all the experiment data. While there were some scenarios where the concurrent DPLL beats the sequential algorithm, it was not enough to expend the resources for that little amount of speed-up. Too much time spent on queue accesses, the location of the solution, and miscellaneous overhead were identified as the reason for little speed-up.

11.4 Further Discussion

This paper has so far discussed a minority of attempts to achieving speed-up. There were many other approaches that proved to be unsuccessful. We experimented in switching the DPLL guesses. Observe in Figure 3, that the root of the tree is 1. What if it was 0, would this change anything? The short answer to this is no, but not conclusive.

In addition, we also attempted to have the sequential and concurrent DPLL algorithms attempt to obtain all the solutions. The logic behind this was that the concurrent algorithms would gain an edge on the solution location issue. In order to modify the algorithms to do this, we had them all return false, even if they found a solution. They would just count the number of times they found a solution. When they counted up to 2 times the number of solutions, they would return true and the algorithms would finish.

Why count to 2 time the number of solutions? By the nature of the DPLL algorithm, it does not notice it has found a solution until it does $n + 1$ calls to DPLL, with n being the number of variables. It does an extra call, notices it has no more clauses to check, and now can return TRUE. If it instead returns FALSE, it still will continue to search deeper into the tree. Actually, it only makes one extra call, but still has no more clauses so it will count once again that a solution has been found, even though this solution has been accounted for. All in all, the DPLL algorithms will make $n + 2$ calls to DPLL before it back tracks higher up into the tree.

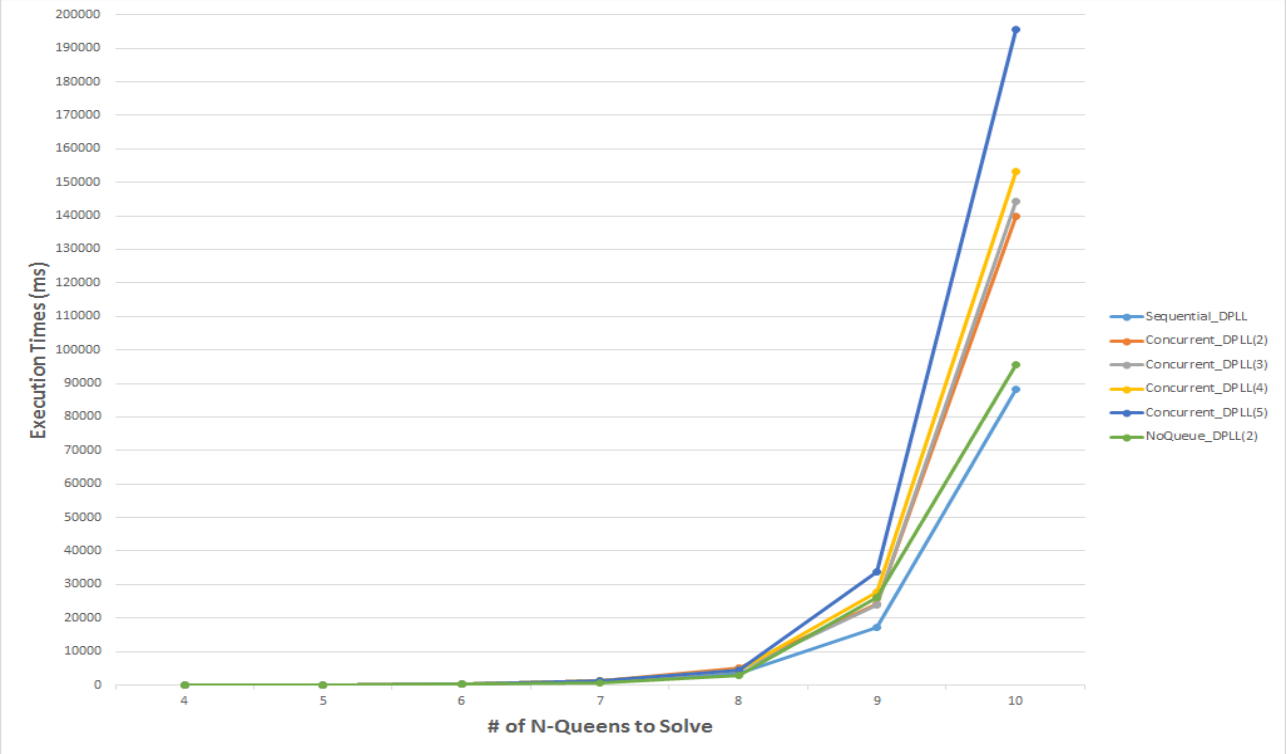

Figure 5: Graph of all experiment results where algorithms solve for all solutions.

Figure 5 shows execution times for all DPLL algorithms when solving for all solutions. The results are a little baffling. There are many cases where the sequential algorithm proved to be the quickest. The only explanation we can think of for this is from overhead. This result was not expected and the experiment was put together quite quickly, hence being added to the further discussion section.

References

[1] (2014) Boolean satisfiability problem. [Online]. Available: http://en.wikipedia.org/wiki/Booleansatisfiabilityproblem

[2] (2014) Queens problem. [Online]. Available: http://mathworld.wolfram.com/QueensProblem.html

[3] (2014) Moder sat solvers. [Online]. Available: https://courses.cs.washington.edu/courses/csep573/11wi/lectures/ashish-satsolvers.pdf

[4] Bradley and Manna, *The Calculus of Computation.* Morgan Kaufmann Publishers, Inc., 2007.

[5] (2014) Dpll algorithm. [Online]. Available: http://www.dis.uniroma1.it/ liberato/ar/dpll/dpll.html

[6] (2014) Concurrentqueue class. [Online]. Available:
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html