School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A Project Report: The N-Queens Problem

Rylan Perumal, 1396469, Computer Science Honours
Tasneem Abed, 1408535, Big Data Analytics Honours

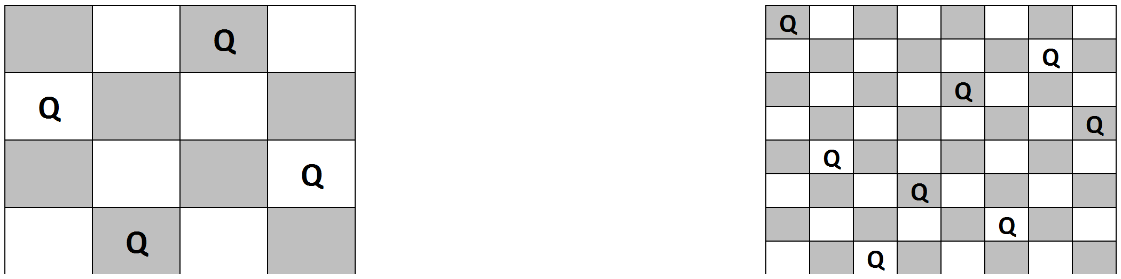
June 22, 2019

1 Introduction

The N-Queens problem is to place N Queens on an $N \times N$ chessboard so that no Queens attack each other. Attacking Queens are in the same row, column or diagonal to each other. Further, we want to determine how many possible board configurations exist. The N-Queens problem was originally proposed by chess player Max Bessel in 1848 as the 8-Queens problem. The problem serves as a benchmark for algorithms that solve constraint satisfaction problems (CSPs). Despite direct applications of the problem being limited, studies of the problem produced useful results for dealing with large CSPs and continues to be used as a base for developing new solutions to CSPs [Sosic 1994].

On an $N \times N$ board there are N^2 possible placements for the first Queen, $N^2 - 1$ for the second Queen, $N^2 - 2$ for the third Queen and so on. Thus a naive (brute force) approach would be to try all possible placements of each Queen and then select valid solutions. As N increases this becomes an extremely large number with $N = 10$, having 1.73e13 possible solutions and $N = 21$, having 314,666,222,712 possible solutions. A good approach to this problem is to use a backtracking algorithm which places Queens from the leftmost column to the right. The backtracking algorithm is a tree search algorithm (DFS - depth first search).

The aim of this project is to parallelize the backtracking algorithm. Solving this problem using a parallelized approach offers the potential to be extremely efficient. CUDA (Compute Unified Device Architecture) and MPI (Message Passing Interface) are platforms used to parallelize the algorithm. In section 2, we discuss the methodology, which explains the backtracking algorithm, the serial implementation, the idea to parallelize the algorithm and the MPI and CUDA implementations. In section 3 we discuss the experiment setup as well as the hardware specifications. Section 4 presents the results from the the experiments that were run. A discussion and analysis of the results is included in section 5 which is followed by a brief discussion of the challenges and improvements in section 6 and our final conclusions in section 7.


(a) A solution to the 4-Queens Problem

(b) A solution to the 8-Queens Problem

Figure 1: 2D visualization of the board

2 Methodology

In the following subsections we discuss the backtracking algorithm [1] used, the serial implementation, the approach to parallelize the problem and the CUDA and MPI implementation using the parallelization approach and backtracking.

2.1 Backtracking Algorithm

- Start at the left most column.
- If all the Queens are placed, return true.
- For each column, try to place Queens in all rows.
- For each row:
  1. Check to see if the placement of the Queen is valid.
  2. If valid, mark the corresponding column and row as part of the solution and check if placing Queens from here leads to a solution.
  3. If placing a Queen in this position leads to a solution, return true.
  4. If the placement of the queen does not lead to a solution, remove the Queen and backtrack. (Go to step 1 and try other possible rows in the column.)
- If all rows have been tested and leads to no solution, the algorithm returns false to trigger backtracking.

2.2 N-Queens Problem in Serial

2.2.1 Representation

The board is stored as an array of length $1 \times N$, where the index of the array element corresponds to the column and the value of the array element corresponds to the row at which a Queen is placed on the $2D$ board [2]. 2 below is the 1D representation of the board that corresponds to the 2D representation of the solution in 1 . Storing the board as a 1D array saves space for large N.

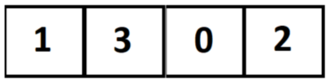


Figure 2: 1D representation of the board

2.2.2 Implementation

The serial algorithm is recursive and places Queens sequentially from left to right ensuring that no Queens attack each other. The solution count is incremented every time a valid solution is found and the total solution count and overall running time is displayed. This implementation is simple and intuitive. Using the Backtracking algorithm over a brute force approach reduces computation time by reducing the possible number of arrangements of each Queen.

[1] `https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/`
[2] `http://www.drdobbs.com/jvm/optimal-queens/184406068`

2.3 Parallelization

For a partial board configuration where the number of Queens placed $< N$, we want to parallelize the computation of the remainder of the possible solutions from this point in the board. That is, we want to partially compute the solution up to a certain depth in serial, where each partial board configuration is valid and unique, up to the specified depth then compute the rest of the board in parallel.

2.4 N-Queens Problem in MPI

2.4.1 Representation

The board is stored as an array of length $1 \times N$, where the index of the array element corresponds to the column and the value of the array element corresponds to the row at which a Queen is placed on the $2D$ board as seen in Figure 2. This saves space when dealing with large N.

2.4.2 Implementation

Based on the aforementioned parallel idea, for each process we set the first element in the array to the process number i.e. array[0] = process number. Each process will have a unique initial board configuration in which it will find solutions for each configuration in parallel. Each worker process computes all valid solutions for its unique board configuration and sends the solution count to the master process. The master process computes the total sum by summing the solution count sent from each worker process and displays it with the running time. The master process is not involved in any computation of solutions for the N-Queens problem. Due to this, the implementation forces the number of processes used, which is user defined, to be $N+1$.

Based on our above mentioned parallel idea, this approach seemed effective and plausible as we could assign each process a unique board configuration based on its process number. This allowed possible board configurations to be computed in parallel as the placement of the first Queen at array[0] will always be valid since we place Queens from left to right. Placing Queens in different row positions in the first column creates the possibility of different solutions to be computed.

2.5 N-Queens Problem using CUDA

2.6 Algorithm

This algorithm is proposed by Somers [2002]. For a new Queen, a placing bit-mask is used to select a free place. After placing a Queen, three lists of masks which correspond to the occupied columns, positive and negative diagonals in the $N \times N$ board are updated. For the next row, the positive diagonal is shifted one to the right and the negative diagonal is shifted one to the left. The placing bit-mask is then recalculated and a new Queen will be placed. If no place is available, the algorithm then backtracks to the most recently placed Queen, removes it, and places it in the next valid position. If all Queens are set then the total solution counter is incremented.

2.6.1 Implementation

Using the aforementioned parallel idea, partial solutions up to a certain depth is calculated in serial (this depth is user specified). These partial solutions (board configurations) are then passed to the GPU where the number of threads are equal to the number of valid board configurations up to the specified depth. Each thread will compute the possible solutions for their specific board configuration. The total solution count, kernel run time and overall run time are displayed when the program is complete.

This implementation does not use recursion, it uses a stack which is stored in shared memory on the device. CUDA does not handle recursion well, especially when the depth of the recursion is unknown. An advantage of using the bitmasks is that it consumes little memory. This algorithm is well known for its use in successful implementations of the N-Queens problem.

3 Experiment Setup

All implementations were tested for increasing values of N, where $N = [5, 6, ..., 18]$. The board sizes of $N > 18$ takes significantly longer than $N \leq 18$ to run, especially the serial implementation, thus we have not included it in our experiments.

3.1 MPI Experiments

Our implementation forces the number of processes used to be $N + 1$. Thus we further varied our tests by testing it on different nodes. The implementation was tested on 1, 2, 4 and 8 nodes. Where the number of processes is evenly distributed between nodes.

3.2 CUDA Experiments

For the CUDA implementation we varied the number of threads tested as well as the depth for different values of N. The different depths that were used were 2, 4 and 5. However, we did not use depth 4 to test $N = 5$ and depth 5 to test $N = 5, 6$ as it is not beneficial to use a parallel implementation to solve the board when $(N - D) \leq 1$, where D is the depth, as this is almost equivalent to running it in serial. The different number of threads that were tested were 32, 64 and 128.

3.3 Hardware Specifications

The serial and CUDA implementations were run on the computers in MSL which have the following specifications:

- Memory: 15.6 Gib
- CPU: Intel Core i7-7700 CPU @ 3.60GHz x 8
- GPU: GeForce GTX 1060 6GB/PCIe/SSE2
  - Memory Clock Rate: 4.004 GHz
  - Theoretical Peak FLOPS: 4.375 TFLOPS
  - Peak Memory Bandwidth: 192.192 GB/s
  - Memory Bus Width: 192 bits
  - Compute Capability: 6.1

The MPI implementation was run on the MSL cluster where each node has the following specifications:

- CPU: Intel Core i7-7700 CPU @ 3.60GHz x 8
- CPU MHz: 4000.048
- Cache size: 8192 KB

4 Results

4.1 Results of serial implementation

N	Serial implementation time(s)
5	0.000199
6	0.000352
7	0.000506
8	0.000328
9	0.001392
10	0.007353
11	0.035007
12	0.194514
13	1.150302
14	7.455840
15	52.429065
16	380.272561
17	2925.338955
18	23937.846821

Table 1: Table showing running times for the serial implementation in seconds

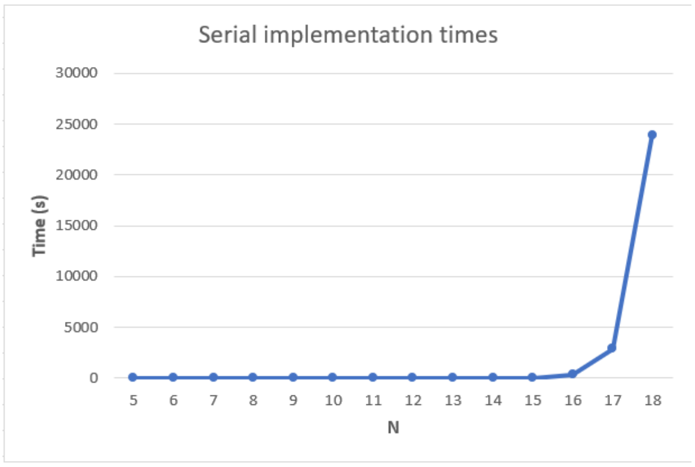


Figure 3: Graph of run times of the serial implementation

4.2 Results of MPI implementation

N	Number of Processors	MPI 1 node	MPI 2 nodes	MPI 4 nodes	MPI 8 nodes
5	6	0.000180	0.000487	0.000417	0.000684
6	7	0.000339	0.000209	0.00054	0.000542
7	8	0.015996	0.000701	0.000976	0.000377
8	9	0.015619	0.000953	0.000642	0.000383
9	10	0.026665	0.000951	0.001026	0.000958
10	11	0.028596	0.001107	0.001794	0.001678
11	12	0.029073	0.049032	0.060264	0.006148
12	13	0.095830	0.061920	0.068298	0.028306
13	14	0.478673	0.312390	0.165198	0.163430
14	15	2.550657	1.637908	1.126333	1.003531
15	16	17.762841	9.477976	6.560924	4.074749
16	17	106.234332	56.360553	84.550315	44.579891
17	18	822.861481	812.614667	373.463048	199.685147
18	19	6731.086641	5680.123907	3314.952049	1556.092065

Table 2: Table showing running times in seconds for the MPI implementation running on the cluster with varying number of nodes.

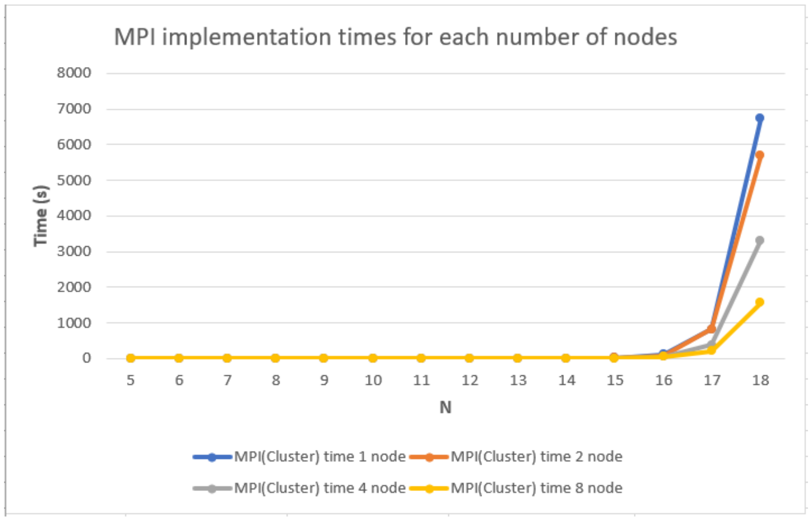


Figure 4: Graph of MPI implementation running times with varying number of nodes

4.3 Results of CUDA implementation

N	Depth 2 Total time 32 Threads	Kernel time 32 Threads	Total time 64 Threads	Kernel time 64 Threads	Total time 128 Threads	Kernel time 128 Threads
5	0,172828	0,000038	0,152848	0,000034	0,14971	0,000034
6	0,147136	0,00004	0,159672	0,000038	0,146482	0,000038
7	0,147426	0,000054	0,159672	0,000056	0,147742	0,000048
8	0,159924	0,00009	0,150118	0,000078	0,166808	0,000078
9	0,152468	0,000202	0,14872	0,000168	0,14687	0,00017
10	0,155036	0,000756	0,147586	0,000612	0,150608	0,000608
11	0,147522	0,001986	0,149882	0,001656	0,151368	0,001648
12	0,158468	0,009328	0,157386	0,007054	0,1541	0,007054
13	0,187588	0,038668	0,180336	0,032056	0,17788	0,032248
14	0,364354	0,219422	0,328864	0,180926	0,333754	0,180104
15	1,017796	0,870738	1,015184	0,867626	1,010948	0,862954
16	5,35272	5,201014	5,44816	5,297764	5,44536	5,29287
17	31,124702	31,007954	31,798656	31,646472	31,729712	31,570262
18	262,167062	261,940454	287,422922	287,200594	287,812812	287,579532

Table 3: Table showing running times in seconds for the CUDA implementation with depth 2.

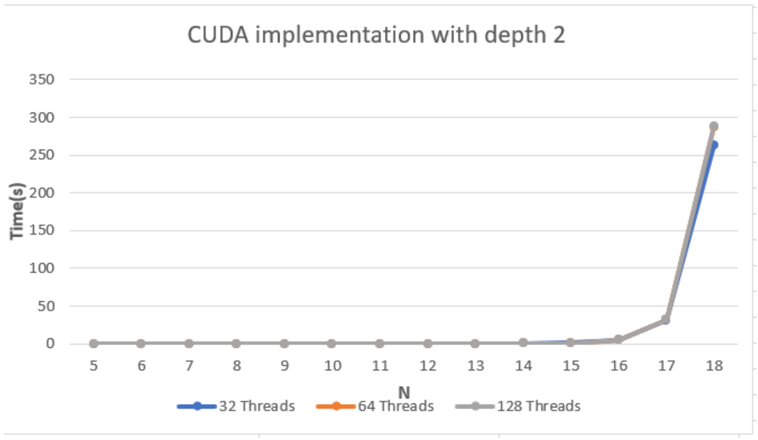


Figure 5: Graph of CUDA implementation with depth 2

N	Depth 4					
	Total time 32 Threads	Kernel time 32 Threads	Total time 64 Threads	Kernel time 64 Threads	Total time 128 Threads	Kernel time 128 Threads
5	-	-	-	-	-	-
6	0,146418	0,000034	0,146948	0,000034	0,14877	0,000034
7	0,14802	0,000034	0,146666	0,000036	0,153918	0,000036
8	0,146936	0,00004	0,146038	0,000042	0,151488	0,00004
9	0,146456	0,000052	0,149184	0,000054	0,167658	0,00005
10	0,148146	0,000082	0,149462	0,000084	0,147386	0,000084
11	0,149184	0,000156	0,146996	0,00016	0,147322	0,000156
12	0,151998	0,000546	0,1493	0,000552	0,15548	0,00054
13	0,151678	0,001446	0,150796	0,001458	0,153356	0,00145
14	0,158356	0,006164	0,1511	0,00615	0,157012	0,006062
15	0,183884	0,03348	0,183496	0,034386	0,182944	0,035826
16	0,432918	0,272578	0,454384	0,305638	0,456126	0,304104
17	1,505164	1,353982	1,550262	1,400466	1,746034	1,58200
18	10,59515	10,437462	10,691886	11,192106	11,696096	11,544392

Table 4: Table showing running times in seconds for the CUDA implementation with depth 4.

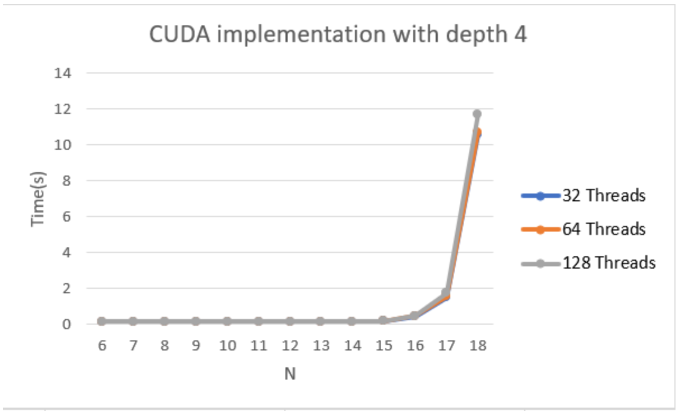


Figure 6: Graph of CUDA implementation with depth 4

N	Depth 5					
	Total time 32 Threads	Kernel time 32 Threads	Total time 64 Threads	Kernel time 64 Threads	Total time 128 Threads	Kernel time 128 Threads
5	-	-	-	-	-	-
6	-	-	-	-	-	-
7	0.11856	0.000038	0.126312	0.000036	0.150388	0.000039
8	0.1191	0.000038	0.14323	0.000034	0.12263	0.000035
9	0.142106	0.000047	0.117036	0.000043	0.146208	0.000047
10	0.146158	0.000063	0.118588	0.00005	0.123014	0.000051
11	0.12407	0.000099	0.138562	0.000086	0.123198	0.000087
12	0.13768	0.000253	0.120028	0.000238	0.12351	0.000247
13	0.12312	0.002256	0.146536	0.001806	0.122914	0.001912
14	0.12752	0.005117	0.125498	0.004205	0.126294	0.004289
15	0.158376	0.034563	0.180742	0.028584	0.15509	0.03103
16	0.35271	0.196882	0.290508	0.165073	0.323512	0.172258
17	1.764328	1.609836	1.7579	1.629159	1.919682	1.789604
18	8.35156	8.196085	8.755638	8.619905	10.18022	10.018417

Table 5: Table showing running times in seconds for the CUDA implementation with depth 5.

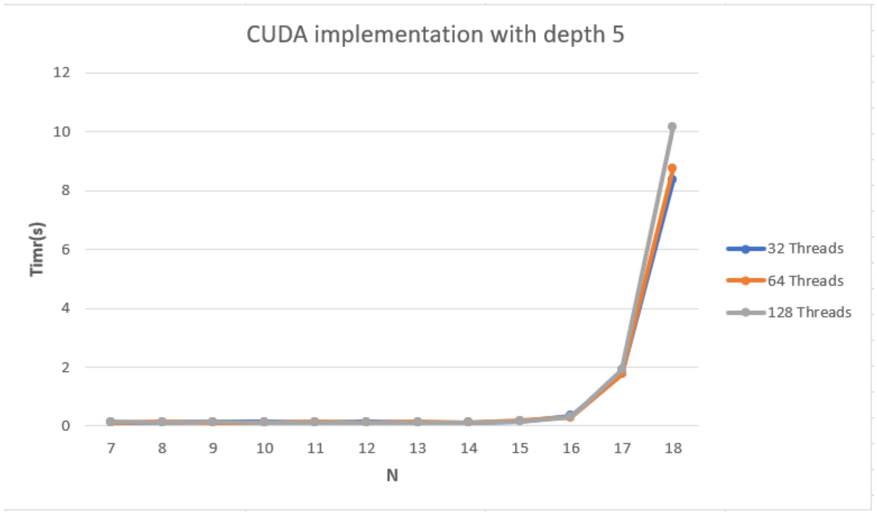
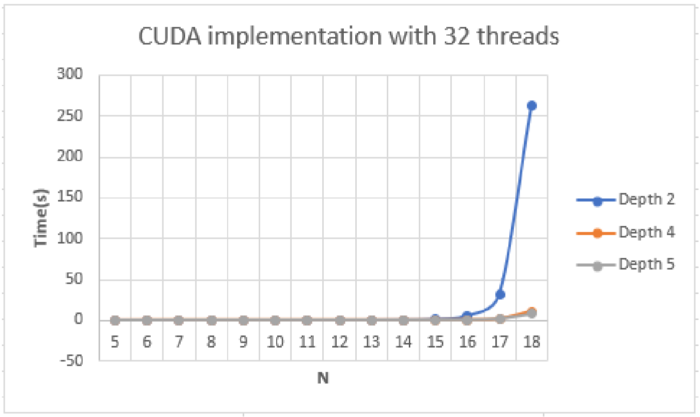
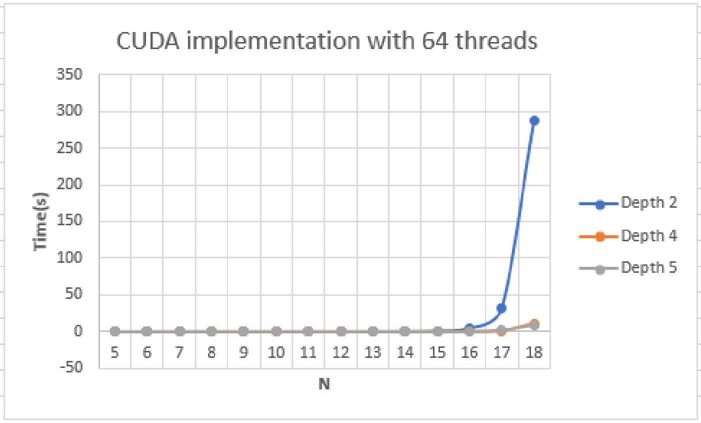


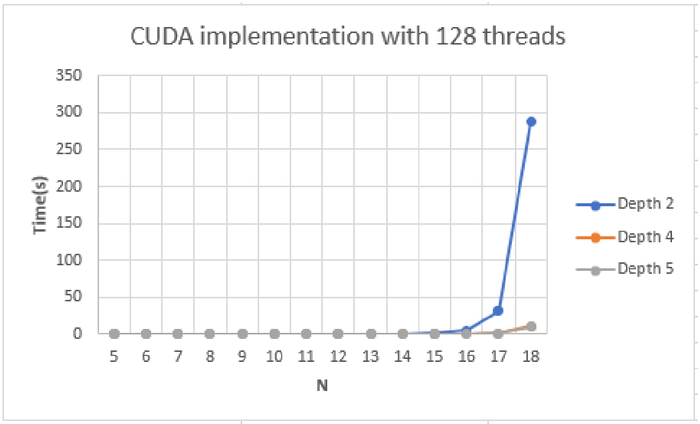
Figure 7: Graph of CUDA implementation with depth 5



(a) Graph of different depths for 32 threads



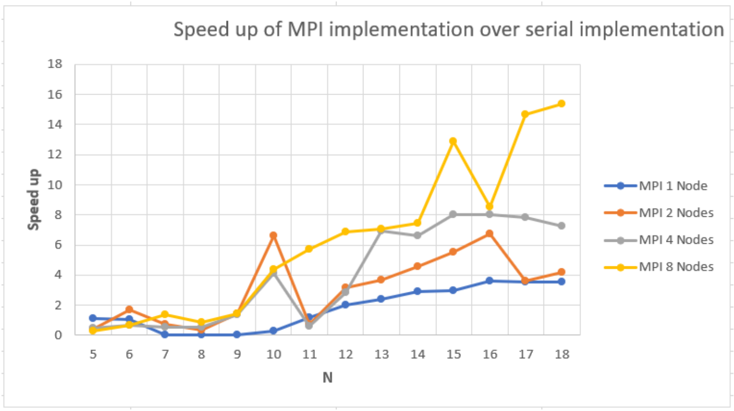
(b) Graph of different depths for 64 threads



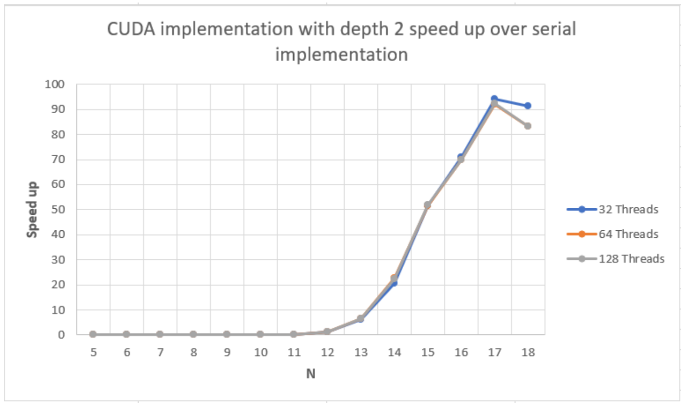
(c) Graph of different depths for 128 threads

Figure 8: Graphs of varying depth per threads

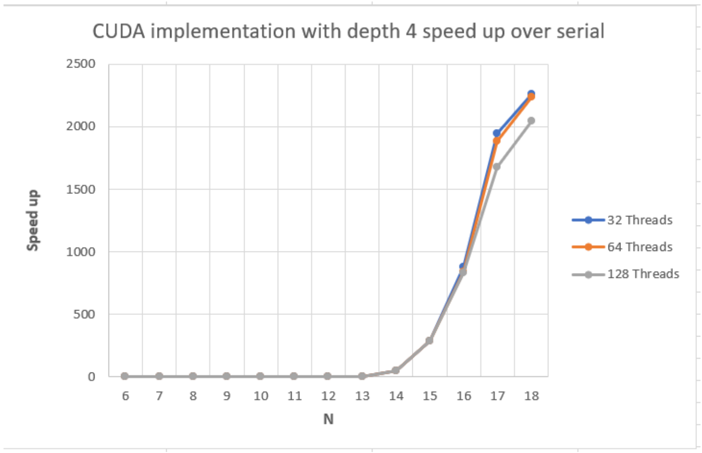
4.4 Speed up



(a) Speed up of MPI over serial implementation



(b) Speed up of CUDA with depth 2 over serial implementation



(c) Speed up of CUDA with depth 4 over serial implementation

Figure 9: Graphs of varying depth per threads

4.5 Comparison

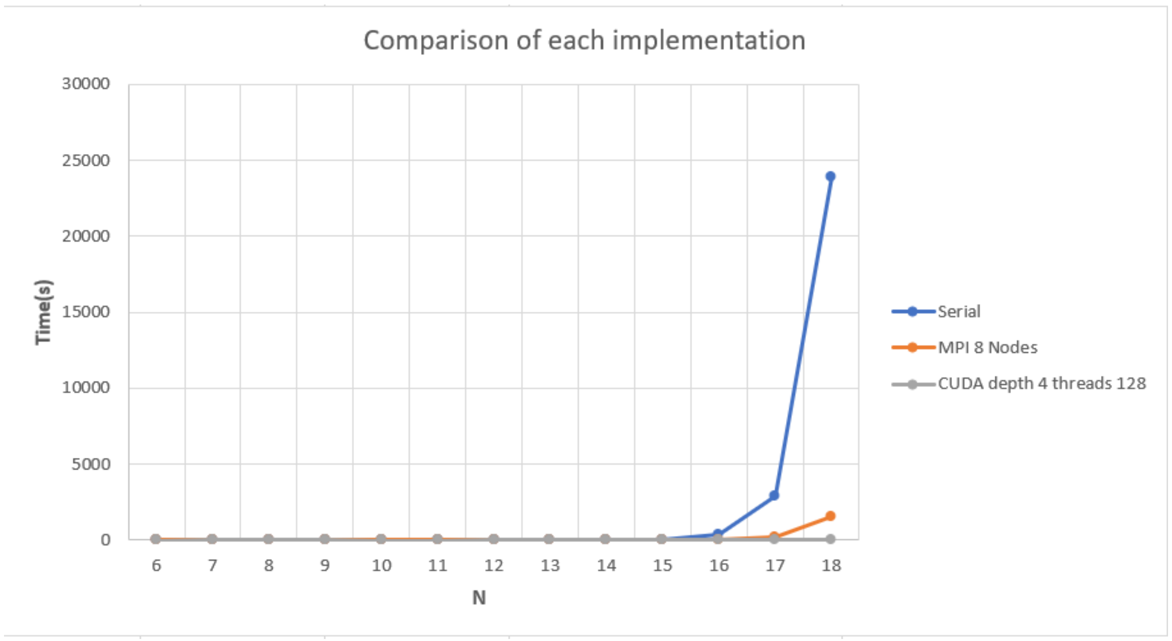
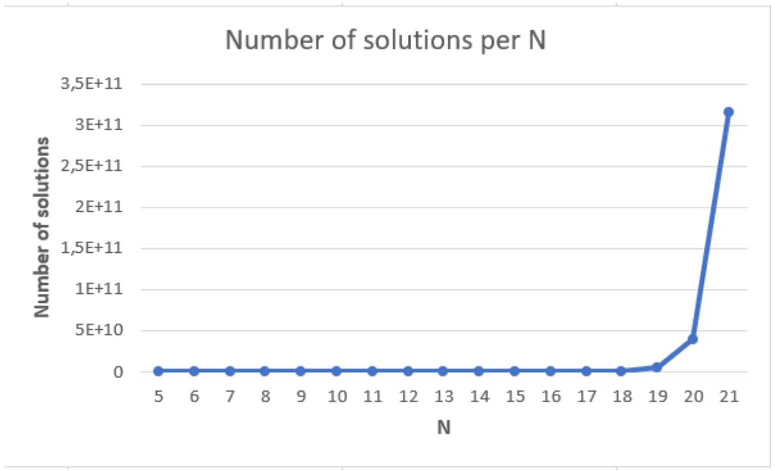


Figure 10: Comparing all 3 implementations using configurations of parallel approaches with best results

4.6 Number of solutions

N	Number of solutions
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512
17	95815104
18	666090624
19	4968057848
20	39029188884
21	314666222712

(a) Table of number of solutions per N



(b) Graph showing increase in number of solutions with increasing N

Figure 11: Number of valid solutions

5 Discussion

From the results we can see that a parallel approach to solving the N-Queens problem is very efficient compared to the serial implementation. The results obtained from our MPI and CUDA implementation improves drastically with respect to time. CUDA overall presented the best overall run time. This can be seen from Figure 10

In Table 1 we can see that at approximately $N = 13$ the jumps in time for each subsequent N gets much larger. For the MPI implementation, 2 shows that the jumps in time start to get large at approximately $N = 15$. Similarly, the CUDA implementation sees larger time jumps from approximately $N = 16$ with depth 2, $N = 17$ with depth 4 and 5. The significance of the larger time jumps happening at larger N values is that it further shows that the parallel implementations carry noteworthy time advantages.

The number of nodes used to run the MPI implementation has a significant effect on the total running time. This is because the more nodes used, the more spread out the processes are and thus the less the processes need to compete for computational resources. For example, if we have 16 processes and 2 nodes, each node will get 8 processes which will compete amongst each other whereas if there were 8 nodes, only 2 processes will need to compete on each node. Figure 4 shows that for an 18×18 board, using 1 node takes about 6700 seconds where using 8 takes about 1550 seconds which is about 4 times faster.

The depth at which the partial solution is computed for the CUDA implementation is crucial. Looking at Figure 8, we can see that as we increase the depth the time taken to find solutions decreases significantly.

Varying the number of threads used for CUDA is not beneficial towards the run time of the algorithm. This can be seen from Figures 5, 6 and 7.

From Figure 9(a), we can see that as we increase the number of nodes used, the speed up increases. This confirms solving the problem in parallel is favourable. However, it is important to note that the speedup fluctuates. This could be due to external factors such as latency issues with the cluster.

From Figures 9(b) and (c) we observe a significant speed up by the CUDA implementation. Furthermore, with an increase in depth, the speed up is drastically improved. Using CUDA as a parallel approach is most efficient in solving this problem with respect to time.

5.1 Time Complexity

5.1.1 Serial

Based on the recursive implementation with backtracking the theoretical time complexity is $O(N!)$ where N is the number of Queens to be placed. From our results in Figure 3 this can be seen for larger values of N by the shape of the graph, the algorithm shows $O(N!)$ behaviour.

5.1.2 MPI

Based on our algorithm and our parallelized approach, the theoretical time complexity is $O(\frac{N!}{P})$ where P is the number of processes used in the computation (excluding the master). From our results in Table 2 and Figure 4 it can be seen that as we increase the number of nodes, the run time decreases. As the number of processes spread out across nodes the less they have to compete for computational resources thus leading to the reduction in time.

5.1.3 CUDA

Based on the algorithm used and our parallelized approach, the theoretical time complexity is $O(D! + \frac{(N-D)!}{B_{partial}})$, where D is the depth to which we partially solve the board before we pass it to the kernel. $B_{partial}$ is the number of partial board configurations prior to the kernel call, which is then parallelized by giving each thread its own partial board to find possible solutions. The first term is the complexity of the serial computation up to the depth. Based on the results in Tables 3, 4, 5 and Figures 5, 6, 7 we can see a significant reduction in time, this is due to using individual threads to compute each partial board.

6 Challenges and improvements

Our MPI implementation forces the number of processes used to be $N+1$. This approach does not generalize well, especially when the number of processes are limited or if we have an excess of nodes which are not being utilized. An improvement to this approach would be to specify an arbitrary number of processes which will be able to solve any $N \times N$ board. Since the master process is not involved in computing solutions and its only job is to sum the total number of solutions, it is sitting idle while the worker processes compute solutions. A further improvement would be to have the master process compute solutions as well as perform a reduction to calculate the total number of solutions. This improvement will change the way the number of processes is specified.

The CUDA implementation uses a bit mask to place a new Queen and then stores the occupied columns, positive and negative diagonals in lists which are used to check if there are any attacking Queens when placing a new one. The concept of using a bit-mask and the process of storing the above mentioned lists is quite involved. Our implementation closely follows the concepts and optimizations used in Feinbube *et al.* [2010].

7 Conclusion

This problem becomes computationally infeasible to solve in serial as $N > 18$. A parallel approach to solving this problem provides a highly substantial improvement with respect to time. More specifically, the CUDA approach produces the most compelling results and is the recommended approach. The MPI approach used in this project does parallelize the algorithm and produces correct results however it could be improved to be optimal and more generalized.

References

[Feinbube *et al.* 2010] Frank Feinbube, Bernhard Rabe, Martin von Löwis, and Andreas Polze. Nqueens on cuda: optimization issues. In *2010 Ninth International Symposium on Parallel and Distributed Computing*, pages 63–70. IEEE, 2010.

[Somers 2002] J. Somers. The n queens problem - a study in optimization. 2002. `http://jsomers.com/nqueen_demo/nqueens.html`.

[Sosic 1994] Rok Sosic. *A parallel search algorithm for the N-queens problem.* School of Computing and Information Technology, Faculty of Science, 1994.