COMS4040A: High Performance Computing Project 2019

The N-Queens Problem

Rylan Perumal (1396469) and Tasneem Abed (1408535)

University of the Witwatersrand

June 21, 2019

Introduction

Problem: Place N Queens on an $N \times N$ chess board so that no Queens attack each other. Queens that are in the same row, column or diagonal attack each other.

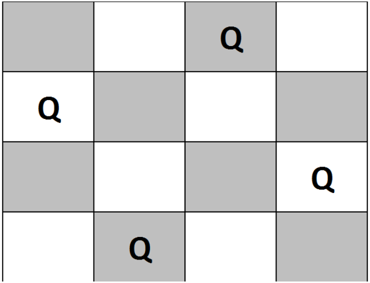

Figure 1: A solution to 4-Queens

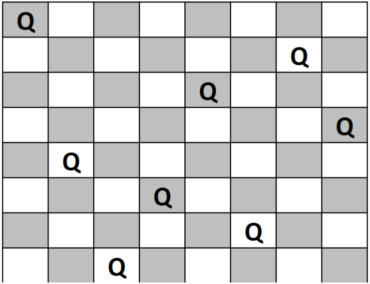

Figure 2: A solution to 8-Queens

Background

- 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]

Backtracking algorithm [1]

- Start at the left most column.
- If all the queens are placed return true.
- For a given column, try to place queens in all rows. Then 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 queen in the above mentioned position leads to a solution, return true.
  4. If the placement of the queen does not lead to a solution, remove the position from the solution and backtrack. (Go to step 1 and try other possible rows in the column.

[1] https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

Backtracking algorithm cont.

- If all rows have been tried and leads to no solution the algorithm returns false to trigger backtracking.

N-Queens problem in serial

Representation

- The board is stored as an array of length $1 \times N$, where the index of the array corresponds to the column and the value of the array at a given index corresponds to the row at which a queen is placed on the $2D$ board.

Implementation

- The algorithm places queens sequentially from left to right ensuring that no two queens attack each other. This is a recursive algorithm which uses backtracking.
- The solution count is incremented every time a valid solution is found.

N-Queens problem in serial

- The total solution count and the overall running time is then displayed.

Reasoning behind implementation [2]

- Intuitive
- Storing the board as a 1D array saves space for large N.
- Using a backtracking algorithm over a brute force reduces computation time by reducing the possible number of arrangements of each queen.

[2] http://www.drdobbs.com/jvm/optimal-queens/184406068

Parallelization

Key Ideas

- For a given partial board configuration where the number of queens placed $< N$, we want to parallelize the computation of the possible solutions from this board configuration.
- That is, we want to compute the partial board configurations up to a certain depth in serial, where each partial board configuration is valid and unique up to the specified depth.
- For each unique partial board configuration, we want to compute the possible solutions in parallel.

N-Queens problem in MPI

Representation

- The board is stored as an array of length $1 \times N$, where the index of the array corresponds to the column and the value of the array at a given index corresponds to the row at which a queen is placed on the $2D$ board.

Implementation

- Based on the aforementioned parallel idea, for each process we set the first position i.e. col[0] = process number. Each process will have an initial unique board configuration in which it will find solutions for each unique configuration in parallel.

N-Queens problem in MPI

- Each worker process computes all valid solutions for its unique board configuration and sends the solution count to the master process.
- The master process then computes the total sum by summing the solution count sent from each worker process and displays it as well as the running time. The master process is not involved in any computation for the N-Queens problem.
- Our implementation forces the number of processes used to be $N + 1$.

N-Queens problem in MPI

Reasoning behind implementation

- Based on our above mentioned parallel idea, this approached 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 col[0] will always be valid since we place queens from left to right.
- Placing queens in different row positions for the first column creates the possibility of a different solutions to be computed.

N-Queens problem in CUDA [3]

Algorithm

- This algorithm is proposed by Somers [2002].
- For a new queen it uses a bit-mask to select a free place.
- After placing a queen it updates three lists of masks which correspond to the occupied columns, positive and negative diagonals in the $N \times N$ board.
- 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.

[3] https://www.academia.edu/18081973/NQueens_on_CUDA_Optimization_Issues

N-Queens problem in CUDA

- If no place is available, the algorithm then backtracks to the most recently placed queen, removes it and places it in a different valid position. If all queens are set then the total solution counter is incremented.

N-Queens problem in CUDA

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 a given board configuration.
- The total solution count, kernel run time and the overall run time are then displays when the program is complete.

N-Queens problem in CUDA

Reasoning behind implementation

- This algorithm 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.
- Consumes little memory with the use of bit-masks.
- This algorithm is well known for its use in successful implementations of the N-Queens problem.

Experiments

Serial and CUDA

These implementations were run on the lab computers with the following specs:

- Memory: 15.6 Gib
- CPU: Intel® Core$^{\text{TM}}$ 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

Experiments

MPI

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

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

N-Queens problem in serial

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

Figure 3: Running times for serial implementation for varying N.

N-Queens problem in serial

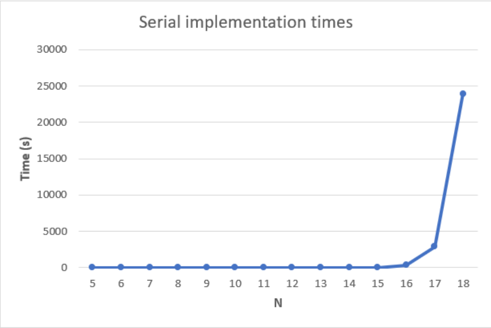

Figure 4: Graph of running times for serial implementation for varying N.

N-Queens problem in serial

Time complexity

- Based on the recursive function with backtracking the theoretical time complexity is $O(N!)$ where N is the number of queens to be placed.
- From our results in Figures 3 and 4 this can be seen for larger values of N, the algorithm shows $O(N!)$ behaviour.

N-Queens problem in MPI

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

Figure 5: Running times in MPI using different number of nodes for varying N.

N-Queens problem in MPI

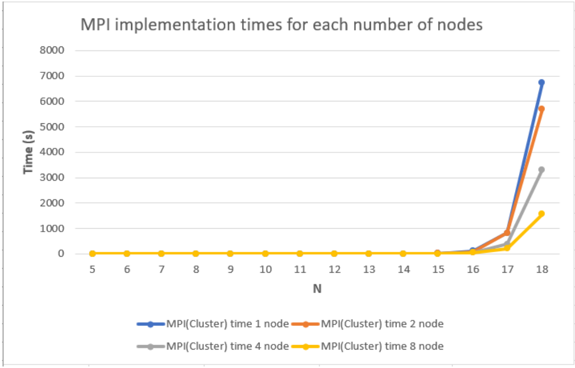

Figure 6: Graph of running times in MPI using different number of nodes for varying N.

N-Queens problem in MPI

Time complexity

- 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). Hence we specify $N+1$ processes at compile time.
- From our results in Figures 5 and 6 it can be seen that as we increase the number of nodes, the run time decreases. As the number of processes are more spread out across nodes the less they have to compete for computational resources.

N-Queens problem in MPI

Speed up

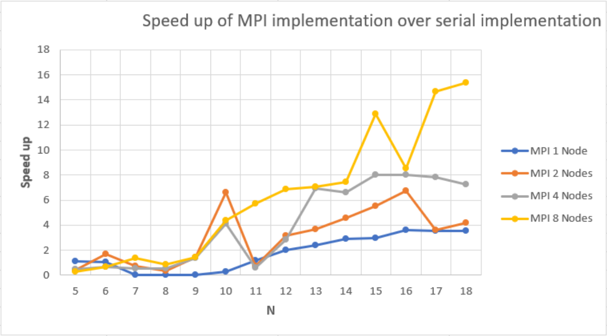

Figure 7: MPI speed up for varying N with increasing number of nodes.

N-Queens problem in MPI

Speed up

- From Figure 7 we can see that increasing the number of nodes increases the speed up. For $N = 18$, the speed up using the MPI implementation with eight nodes compared to the serial implementation is approximately 16. Overall speed up increases significantly when using more nodes.

N-Queens problem in CUDA

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

Figure 8: CUDA running times at depth 2, varying number of threads and N.

N-Queens problem in CUDA

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

Figure 9: CUDA running times at depth 4, varying number of threads and N.

N-Queens problem in CUDA

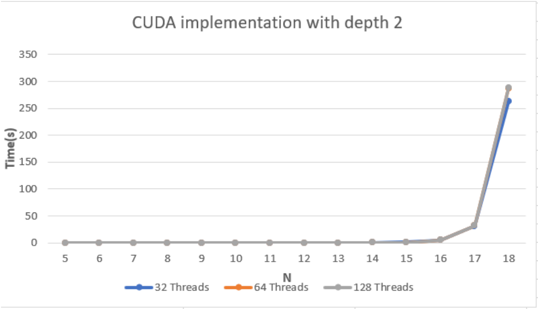

Figure 10: Graph of CUDA running times for depth 2.

N-Queens problem in CUDA

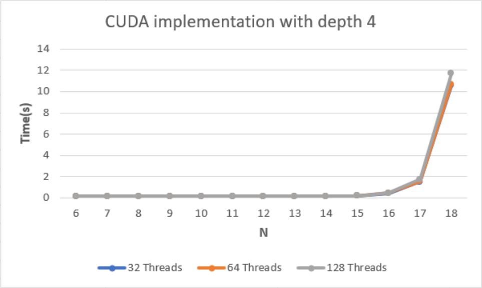

Figure 11: Graph of CUDA running times for depth 4.

N-Queens problem in CUDA

Time Complexity

- 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.
- Based on the results in Figures 10 and 11. We can see a significant reduction in time, this is due to using individual threads to compute each partial board.

N-Queens problem in CUDA

Speed up

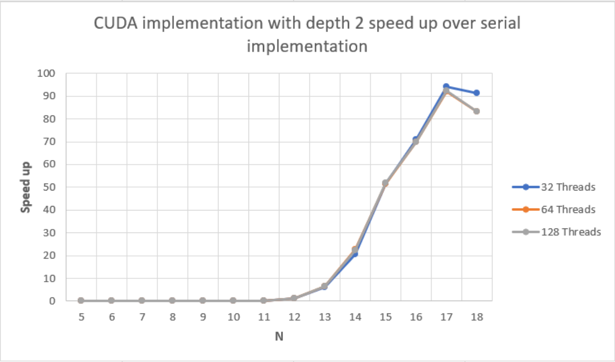

Figure 12: Speed up of CUDA at depth 2 for varying number of threads

N-Queens problem in CUDA

Speed up

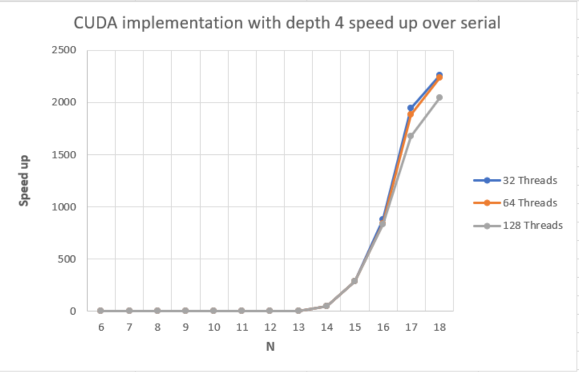

Figure 13: Speed up of CUDA at depth 4 for varying number of threads

N-Queens problem in CUDA

Speed up

- From Figures 11 and 13 we observe significant speed up for $N >= 15$. At $N = 18$ all CUDA implementations present large speed up.

Comparison between implementations

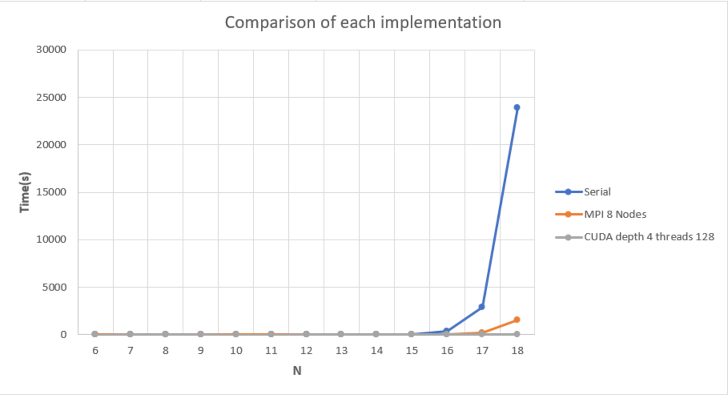

Figure 14: Comparison of all 3 optimal implementation times

Conclusions

Serial

- Performance for large N is poor
- Not a recommended approach to solve this problem for large N and by extension any CSPs.

MPI

- Presents a favourable approach to solving the N-Queens problem
- Provides a noteworthy speed up over the serial implementation.

CUDA

- The performance of the CUDA implementation is exceptional overall
- Varying the depth at which the partial solution is computed affects the overall computation time
- Increasing the number of threads used is not necessarily beneficial. This is because the more threads per block, the less shared memory per thread is available which results in slower computation

Conclusion contd.

- From the comparison graph in Figure 14, we can see that both parallel approaches present major time benefits over the serial implementation.
- The MPI implementation is much simpler and produces favourable results
- CUDA outperforms all other implementations despite its involved implementation.

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.