A Comparative Study on N-Queens Parallelization

Haoxu Huang hh2740@nyu.edu Lubin Sun ls6211@nyu.edu
Wenbo Bao wb2128@nyu.edu Yuhao Fan yf2218@nyu.edu

***Abstract*—The N-Queens problem is a classic combinatorial problem that requires placing N non-attacking queens on an $N \times N$ chessboard. This problem has garnered significant interest due to its complexity of being NP-complete and its applicability in solving real-world problems, such as scheduling, constraint satisfaction, and optimization. Solving the N-Queens problem for large N can be computationally expensive, necessitating efficient algorithms to explore the search space. With the advent of multi-core architectures, parallel computing provides a promising avenue for accelerating the N-Queens problem's solutions. In this study, we present parallel approaches (Brute Force, Backtracking and Genetic Algorithm (GA)) for solving the N-Queens problem using OpenMP based on different original sequential solutions. Our experimental results show that parallelizing with OpenMP can effectively reduce the execution time given N is large. Additionally, we show that Genetic Algorithm is the best approach after parallelization if N is large and we only want to find a few unique solutions. Backtracking Algorithm with parallelization is still effective if we want to find all solutions while Genetic Algorithm is not a good fit for the exhaustive search. Our code is available at https://github.com/bao-wenbo/multicore_project**

I. Introduction

N-Queens is a well-known combinatorial puzzle originating from the eight queens puzzle published by Max Bezzel in 1848. This classic problem has not only captivated researchers with its intriguing combinatorial nature but has also found applications in various real-world scenarios. As the size of the problem (N) increases, the complexity of identifying all valid solutions grows exponentially, prompting the development of efficient algorithms to navigate the search space and minimize computational time.

Over the years, researchers have proposed diverse algorithms and techniques to tackle the N-Queens problem, including backtracking, branch-and-bound, genetic algorithms, and local search methods. The performance of solving N-Queens using differnt algorithms sequentially is widely studied [4], [6], [9]. With the advent of multi-core processors and parallel computing architectures, parallelism has emerged as a compelling approach to further expedite the resolution of the N-Queens problem. Parallel computing allows simultaneous exploration of the solution space, effectively harnessing the computational capabilities of multi-core systems.

Compared with the brute-force algorithm, the backtracking algorithm is a classical method to solve the N-queens problem. It improves the time complexity of the problem from the n power of n to the factorial of n. And many improvements have been proposed on its basis.

In this paper, we explored how parallelism implemented with OpenMP can benefit different algorithms on solving N-Queens problem, where our approaches can be potentially used on solving other similar yet complex combinatorial problems.

Although GA is not good fit for solving all possible solutions of N-Queens, it can be used to find one valid solution much faster than other methods. In general, people found GA can be very efficient on solving N-Queens problem when N is very large (e.g. $N > 100$) in comparison to traditional backtracking deterministic methods. Hence, we are exploring how can we at most further reduce the time cost of running GA on N-Queens problem with OpenMP. Even though N-Queens problem itself has no much practical use, GA can be widely used on other problems such as Function Optimization, Machine Learning Feature Selection, Game Playing and Strategy Optimization and etc. Hence, it would be important to explore the performance improvement of GA on parallel programming in addition to the commonly used brute force and backtracking methods, which can be potentially used in wider appropriate areas.

II. Literature Survey

A. Brute-Force Search

In computer science, the brute-force search, also known as exhaustive search or generate and test, is a versatile problem-solving approach and algorithmic paradigm. It involves systematically enumerating all potential candidates for a solution and verifying if each candidate satisfies the problem's requirements. Although a brute-force search is easy to implement and guaranteed to find a solution if one exists, its implementation costs are directly related to the number of candidate solutions. This can grow rapidly for many practical problems as the problem size increases, leading to a combinatorial explosion.

Thus, brute-force search is generally employed when the problem size is limited or when problem-specific heuristics can effectively reduce the set of candidate solutions to a manageable size. It is also used when the ease of implementation takes precedence over speed.

The brute-force search technique is widely used for addressing computational problems with moderate complexity. In this method, all possible configurations are generated and evaluated to determine viable solutions. When solving the n-Queens problem, the algorithm starts by placing a queen in the first row of the chessboard and then tries to position queens in subsequent rows while ensuring no two queens can attack each other. The algorithm will output a solution if it finds one; otherwise, it will continue generating all possible configurations until a solution is found or all options have been exhausted. Although the brute-force search can effectively solve the n-Queens problem for smaller values of N, it

becomes computationally impractical for larger values, requiring more efficient algorithms for practical problem-solving. However, one notable advantage of the brute-force search is its ease of parallelization, which maximizes efficiency when using multi-core systems.

B. Backtracking Algorithm

The methodology and motivation for the backtracking algorithm were detailed by Edsger W. Dijkstra in August 1971. Dijkstra described a general method for solving problems like 8-queens that find a new greater set to generate all possible solutions in the original set. [3]

The backtracking algorithm follows a logical and systematic approach. Based on the new set we designed, we have the following algorithm. We start by searching the board from top to bottom and left to right. At each step, if the current choice can satisfy all the requirements by placing queens on the current grid, it is considered a new possible placement. And a new branch is generated for searching on the next row in the future. If not, we backtrack to the previous step and perform a check on the next column until we complete the row.

In the context of a search tree, we can think of the backtracking algorithm as a placement tree, where child nodes are placed in a way that extends from the parent node. And all nodes in the sub-tree have the configuration of the current node. This placement tree allows us to handle all possible cases without duplication. In each new level, we add a new queen to the parent's placement, until we reach the N+1 st layer, which represents the final solution as a leaf node in the tree.

C. Genetic Algorithm

Genetic Algorithm (GA) [8] is a powerful heuristic optimization technique to solve complex optimization system. It was introduced by John Holland in the early 1970s and applied to wide variety of problems since then. The Genetic algorithms is inspired by the process of natural evolution and the its general process is divided into selection, crossover, and mutation operations on a population of candidate solutions to optimize a given objective function (fitness function).

Since the advent of GA and N-Queens, there have been many attempts on solving N-Queens with GA. [2] evaluates the performance of solving N-Queens problem with GA by Satisfiability and Heuristic evaluation methods, where Satisfiability evaluation is performed by converting N-Queens into boolean satisfiability function and Heruistic evaluation is calculating number of conflicts in current population. [7] explores modified GA on solving N-Queens problem, where they introduce advanced mutation operation, which does some accidental changes in the population after crossover operation. [1] presents a global parallel genetic algorithm (GPGA) for sovling N-Queens such that the selections and crossovers process in GA can be run simultaneously.

Although previous literatures have demonstrated the enhancement of modified parallel algorithms compared to their sequential counterparts, a comprehensive examination of the advantages and disadvantages among these algorithms has barely been undertaken. In our project, we conduct extensive experiments on multiple algorithms across various settings, providing a more thorough comparison and analysis.

D. Simulated Annealing

In their seminal paper published in 1983, Kirkpatrick et al. introduced the concept and principles of the simulated annealing algorithm for the first time. They demonstrated its important applications in the field of combinatorial optimization through validations on various practical problems such as the traveling salesman problem and graph coloring problem. [5] Similar to genetic algorithms, simulated annealing is a heuristic algorithm.

The first step is to initialize the chessboard, ensuring that the queens do not duplicate. We use an array of length N to represent the positions of the queens in each column. This ensures that no two queens share the same vertical or horizontal positions. This way, we only need to satisfy the requirement that the queens cannot be on the same diagonal by following the steps below.

The second step is to randomly select two columns and swap their queens. Whether the swap is accepted depends on a function. If the swap could decrease conflicts between the queens, it is a good step that we should swap them. Otherwise, the current temperature determines the probability of accepting the swap. Simulated annealing is based on the annealing process in physics, accepting solutions that move towards worse solutions with a certain probability during the search process. in order to avoid getting stuck in the local optimum and increase the chances of finding the global optimum. Repeat the second step until a chessboard with zero conflicts is found, at which point the chessboard represents a solution we want.

Because Simulated Annealing is a heuristic approach same as GA, we only experiment with GA while leaving Simulated Annealing as an extra literature survey.

III. Proposed Idea

A. Parallel Brute-Force

The Brute-Force code iterates over all possible configurations of queen positions and checks if the configuration is acceptable. If so, it increments the number of solutions found. Since we already know all possible configurations, we could do the easiest parallelization over the for loop. The pseudocode is at Algorithm 1.

Find unique solution is achieved on adding an one addition line to cancal all threads once enough solutions are found.

B. Parallel Backtracking

The Backtracking algorithm is to fill every row with one queen by a deep-first order. If N queens is filled, a solution should be added to the answer. Otherwise, we should traverse every cell in the next row to verify if it is threatened. If there is a cell not threatened, we fill that cell with a new queen and

Algorithm 1 Parallel Brute-Force N-Queens Solver

Initialize $number\_solutions \leftarrow 0$
Initialize n to the number of queens
Initialize $max\_iter \leftarrow n^n$ {the total number of configurations}
Parallelize using $num\_workers$ threads:
for $iter \leftarrow 0$ to $max\_iter - 1$ **do**
    Generate a configuration using $iter$
    Check if the configuration is acceptable (no queens attack each other)
    if the configuration is acceptable **then**
        Atomically increment $number\_solutions$
    end if
end for

then traverse the next row untill all the rows are filled with one queen. To find m unique solution, we add a conditional clause to check if m solutions are found. If so, all the search should return. The pseudocode is at Algorithm 2.

Algorithm 2 Parallel Backtracking N-Queens Solver

Initialize $number\_solutions \leftarrow 0$
Initialize n to the number of queens
Initialize m to the solution that need to be found
Parallelize using $num\_workers$ threads:
Set a unique start setting of a local chess board where row queens has been placed already.
if $number\_solutions >= m$ **then**
    return
end if
if filled N row **then**
    $number\_solutions$++ and return
end if
for col from 0 to $n - 1$ **do**
    if $(row + 1, col)$ is a valid position **then**
        Mark $(row + 1, col)$ as queen
        Search the next row recursively
        Mark $(row + 1, col)$ as non-queen
    end if
end for

C. Genetic Algorithm

GA is generally splitted into five parts and executed iteratively until solution is found, with

- Population: Initialize/Maintain a collection of candidate solutions (chromosomes) that represent potential solutions to the optimization problem. In N-Queens case, the candidates are row indices of the placed queens.

- Fitness function: A function that used to evaluate quality of each candidate solution in the population, which means how well a solution meets the optimization objectives. In N-Queens case, the number of conflicts in each chromosome is calculated.

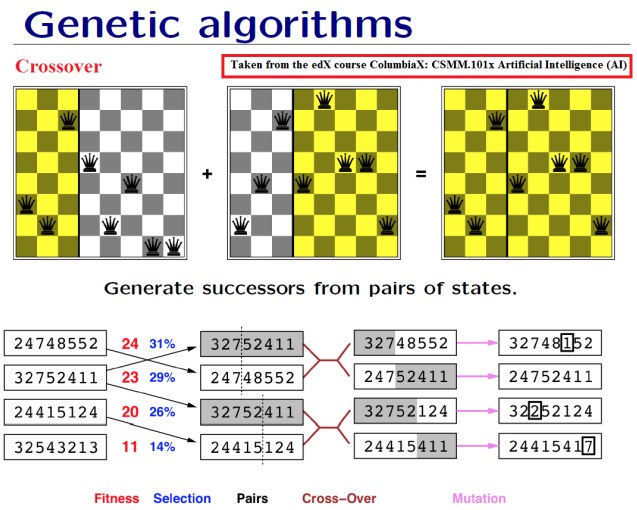

Fig. 1: An Example of One Iteration Genetic Algorithm for Solving N-Queen [1]

- Selection: A process that chooses individuals from the current population based on their fitness, where higher fitness values have higher probability to be selected. This ensures that better-performing solutions are more likely to be chosen for producing the next generation. In N-Queen case, the probability is calculated by $\frac{\text{current non-conflicts in chromosome}}{\text{total non-conflicts in population}}$.

- Crossover: This operation combines two selected chromosomes based on probability to produce offspring solutions by crossing over parts of selected chromosomes. In N-Queen case, two selected sub-parts of the chromosome are crossed.

- Mutation: A random modification applied to the offspring after the crossover operation. Mutation introduces small changes in the offspring's genetic material, helping to maintain diversity in the population and prevent premature convergence to suboptimal solutions. In N-Queen case, an index in the chromosome is replaced with a random new index by chance.

An example of for GA on N-Queens can be visualized in Figure 1. In this example, fitness function is directly calculating the number of non-conflicts for each given chromosome.

D. Find Multiple Solutions in Genetic Algorithm

As the original GA terminates when one solution is found, we propose a solution for finding multiple unique solution for GA within a single run. Specifically, during the Mutation phase, a chromosome exhibiting no conflicts is identified and subsequently removed from the population. The vacant

[1] Credit https://rpubs.com/sandipan/257269

position is then filled with a randomly re-initialized chromosome. Additionally, the identified conflict-free chromosome is appended to an array, which serves as a reference to ensure the uniqueness of subsequent chromosomes. As the experiments show in the later section, this simple modification turns out to work well.

E. Parallel Genetic Algorithm

Because each chromosome in the population is independent, it leaves large space for GA to be optimized, where every step in GA can be made a parallel version. Specifically, the following changes are made in our parallel program to make Genetic Algorithm to be more efficient.

- In the Initial Population process, each chromosome is parallelized to be generated concurrently.
- In the Selection, the loop call for each fitness function is modified to perform in parallel.
- In the Fitness Function, the loop to calculate number of conflicts is parallelized such that the execution can be done in one for loop in parallel. The loops of checking conflicts for horizontally, upward diagonal and downward diagonal are combined into one loop such that it is easier to be parallelized.
- In the Crossover, since the swapping can potentially cause race condition and the loop is too small to outweight the cons of parallelizing overheads, it is not parallelized.
- In the Mutation, all mutations are modified to perform in parallel across all chromosome in the population. For single chromosome, since the mutation is dependent on the current chromosome, it may cause race condition, hence not parallelized.

As a result, we modify the original GA to be highly parallelized and the performance improvement is impressive when N is large as shown in Experiment Section V.

IV. Experimental Setup

In order to exhaustively explore the impact of parallel programming on GA, we perform evaluation on multiple different settings of GA hyperparameters. Specifically, we varied the Number of Queen (N) on very large number in range of $[25, 50, 60, 80, 100]$, Number of Unique Solution to generate in range of $[1, 3, 5]$ and Number of Population to explore in range of $[100, 200, 300, 500]$ with combination of different number of threads in range of $[1, 5, 10, 20]$. Backtracking and brute force search are run in Number of Queen (N) vary in range $[6, 30]$ and number of the $[1, 4, 16, 64, 128, 256, 1024]$ threads.

All our experiments are run on crunchy5 CIMS Machine which runs on four AMD Opteron 6272 processors (2.1 GHz) (64 cores/64 threads).

V. Experiments & Analysis

We show our experimental results in Table I to VI, where Table I and Table II show our experiments on Genetic Algorithm, Table III shows our experiments on Brute-Force and Table IV to VI show our experiments on Backtracking. Some of the sections are marked as '-' as these experiments take too long to finish.

A. Table I Analysis

Due to the optimization of loops (e.g. combing multiple loops to one) in the parallelized version to enhance parallelism, the iteration efficiency is improved, leading to better performance of a single thread compared to the original, unoptimized sequential version, particularly when N or P is large. It is important to note that a larger P does not invariably result in improved performance. Rather, P functions as a hyperparameter, with each N possessing an associated optimal P value. Although the difference is small, we can observe that larger P getting close to the performance of smaller P when number of threads T increases, which indicate the effectiveness of parallelization.

Despite the absence of a notable enhancement in running speed as P increases, it is discernible that values of $P = 200$ or $P = 300$ typically yield marginally higher performance. Moreover, a comprehensive analysis of all N reveals a marked improvement in performance when the number of threads T is less than or equal to 10, while only minimal advancement is observed when $T = 20$. In this particular scenario, it can be postulated that the overhead associated with parallelization, such as synchronization and thread creation, outweight the advantages conferred by the parallelization.

B. Table II Analysis

In accordance with the observations derived from Table II, a more pronounced enhancement in performance is evident when $T \leq 10$. Furthermore, a consistent trend is discerned, indicating that the disparity between the time costs associated with identifying a larger number of solutions and those of a smaller number narrows as T increases. In general, the discrepancy in running speed between finding 1 and 3 distinct solutions remains relatively insignificant, while a greater speedup is observed for finding 5 unique solutions throughout the experimental trials.

C. Table III and IV Analysis

Finding a unique solution is faster than finding all solutions, as the speed is dependent on the magnitude of the total number of solutions. However, the process still scales exponentially. It is interesting to note that although the Crunchy5 machine can only run 64 threads in parallel, in many cases, utilizing more than 64 threads still yields a boost in processing time. This is because breaking the loop into more parts increases the likelihood of stumbling upon a unique solution. Additionally, while not explicitly calculated, the ratio between the time it takes to find one solution and three or five solutions is generally smaller than 1/3 or 1/5. Finding a single solution

Threads	Sequential				T=1				T=5				T=10				T=20			
Population	P=100	P=200	P=300	P=500	P=100	P=200	P=300	P=500	P=100	P=200	P=300	P=500	P=100	P=200	P=300	P=500	P=100	P=200	P=300	P=500
N=25	0.222	0.272	0.291	0.314	0.301	0.310	0.424	0.451	0.185	0.153	0.151	0.172	0.134	0.140	0.118	0.117	0.115	0.120	0.109	0.128
N=50	3.218	3.868	2.751	4.711	3.512	3.655	2.953	4.225	1.527	1.109	1.383	2.561	0.831	0.851	0.551	0.726	0.722	0.811	0.735	0.532
N=60	10.280	5.791	5.665	6.599	6.962	6.599	5.988	5.464	1.634	2.029	2.176	1.802	1.438	1.601	1.523	1.051	1.251	1.158	1.065	1.154
N=80	19.521	27.131	26.852	17.380	18.259	17.335	20.324	19.476	5.710	5.406	6.548	5.578	4.167	4.434	4.425	3.888	2.533	3.111	2.618	2.728
N=100	52.008	47.546	47.561	50.987	42.577	39.748	37.739	43.390	13.072	12.840	12.088	14.137	8.570	7.416	6.406	7.849	6.708	6.819	6.742	6.755

TABLE I: Parallelization Performance on Different Population. The scores are reported by seconds (s) averaged by 5 runs due to the undeterministic behavior of Genetic Algorithm

Threads	Sequential			T=1			T=5			T=10			T=20		
Unique Solutions	1	3	5	1	3	5	1	3	5	1	3	5	1	3	5
N=25	0.272	0.313	0.845	0.310	0.367	0.523	0.153	0.217	0.311	0.153	0.140	0.295	0.140	0.157	0.230
N=50	3.868	4.219	4.890	3.655	3.851	5.595	1.109	1.430	1.633	0.851	1.314	1.789	0.831	0.904	1.114
N=60	5.791	6.110	8.236	6.599	7.215	9.137	1.634	2.606	3.182	1.601	1.970	2.202	1.158	1.332	1.640
N=80	27.131	29.217	35.602	17.335	17.379	24.564	5.710	6.537	7.207	4.434	4.594	5.114	2.533	3.056	3.835
N=100	47.546	74.514	87.947	39.748	54.007	67.256	13.072	14.964	17.074	7.416	8.682	9.331	6.819	7.502	8.784

TABLE II: Parallelization Performance on Finding Different Number of Unique Solutions. Population Size is fixed as 200. The scores are reported by seconds (s) averaged by 5 runs due to the undeterministic behavior of Genetic Algorithm

Threads	T=4			T=16			T=64			T=256			T=1024		
Unique Solutions	1	3	5	1	3	5	1	3	5	1	3	5	1	3	5
N=8	0.0138151	0.0309607	0.0358721	0.00935599	0.0116864	0.0179043	0.00709594	0.00864157	0.00823995	0.0300362	0.0540308	0.0221131	0.0569508	0.0558089	0.0549892
N=9	1.024	1.20282	1.23579	0.0304695	0.0736779	0.0855113	0.00890694	0.0130528	0.0223183	0.0275187	0.0277336	0.0454212	0.0572975	0.0633723	0.0925072
N=10	0.516637	3.14304	3.45418	0.364141	0.522549	1.40726	0.095715	0.158721	0.455668	0.0456686	0.104302	0.323382	0.0669396	0.0810464	0.28823
N=11	14.3747	18.544	44.2884	3.2231	8.76316	11.4321	1.34203	1.39659	2.55687	2.75064	3.4556	4.64202	0.0996671	1.12516	1.61106
N=12	-	-	-	-	-	-	28.4537	29.5791	33.4236	7.57524	11.2642	17.9258	12.9947	13.3658	20.4632

TABLE III: Parallelization Performance on Finding Different Number of Unique Solutions with Brute-Force

Threads	T=4			T=16			T=64		
Unique Solutions	1	3	5	1	3	5	1	3	5
N=20	0.581	0.742	0.758	0.024	0.049	0.059	0.026	0.034	0.065
N=24	2.088	2.107	2.499	0.096	0.084	0.365	0.113	0.090	0.128
N=28	19.905	20.179	20.182	2.781	2.804	3.553	3.198	3.228	3.650
N=29	10.126	10.959	11.139	1.114	3.752	6.434	1.283	1.803	2.255
N=30	-	-	-	-	-	-	6.620	13.857	15.170
N=31	-	-	-	-	-	-	14.969	23.529	26.876
N=32	-	-	-	-	-	-	92.075	89.991	94.99

TABLE IV: Parallelization Performance on Finding Different Number of Unique Solutions with Backtracking

becomes extremely infeasible for larger values of n due to both the problem size and the relatively small number of solutions. In comparison to finding all solutions, finding a unique solution only increases the problem size by 2 within the same amount of time.

Compared to brute force search, finding a unique solution using backtracking is much faster than the corresponding all-solution version. This is because it is much easier to find the first few solutions without excessive backtracking. While the all-solution program runs at n=14, the unique solution program can run at n=30 within the same time frame. However, this version of the backtracking code is still unable to fully benefit from the large number of cores available due to its design.

D. Table V, VI and VII Analysis

In both Tables V and VI, the speedup is determined by dividing the time taken by the single-threaded program, while efficiency is calculated as the speedup divided by the number of cores. Although the brute force search algorithm is easy to parallelize and its speedup closely matches the number of cores utilized, it cannot efficiently handle problem sizes larger than 10 due to its rapid growth, exceeding an exponential rate at $O(n^n)$.

Table VI demonstrates that, while still growing exponentially, the backtracking algorithm outperforms the brute force search, solving problem sizes up to 14 in a relatively short amount of time. However, the recursive nature of most backtracking code makes it challenging to parallelize. In this project, parallelization is limited to the number of possible configurations on the first row of the board, which constrains the code's parallelism. Consequently, the speedup observed is at most $O(n)$ instead of being proportional to the number of threads used, resulting in a lower efficiency compared to the brute force search.

The brute force search only exhibits faster runtime for very small problem sizes, specifically when $n = 6$. The use of

Threads	T=1			T=4			T=16			T=64		
Size/thread	time	speedup	efficiency	time	speedup	efficiency	time	speedup	efficiency	time	speedup	efficiency
N=6	0.00426444	1.0	1.0	0.00141981	3.0	0.75	0.00114161	3.74	0.23	0.00529246	0.81	0.01
N=7	0.092427	1.0	1.0	0.0272598	3.39	0.85	0.00674266	13.71	0.86	0.00999833	9.24	0.14
N=8	2.21995	1.0	1.0	0.58383	3.8	0.95	0.144759	15.34	0.96	0.086089	25.79	0.4
N=9	60.6045	1.0	1.0	15.512	3.91	0.98	4.1246	14.69	0.92	1.69116	35.84	0.56
N=10	-	-	-	-	-	-	119.282	-	-	44.2016	-	-

TABLE V: Parallelization Performance of Brute Force Search on Find All Solutions

Threads	T=1			T=4			T=16			T=64		
Size/thread	time	speedup	efficiency	time	speedup	efficiency	time	speedup	efficiency	time	speedup	efficiency
N=6	0.008	1.0	1.0	0.014	0.57	0.14	0.011	0.73	0.05	0.019	0.42	0.01
N=7	0.008	1.0	1.0	0.011	0.73	0.18	0.013	0.62	0.04	0.018	0.44	0.01
N=8	0.012	1.0	1.0	0.011	1.09	0.27	0.01	1.2	0.07	0.018	0.67	0.01
N=9	0.059	1.0	1.0	0.013	4.54	1.14	0.012	4.92	0.31	0.019	3.11	0.05
N=10	0.101	1.0	1.0	0.031	3.26	0.81	0.016	6.31	0.39	0.026	3.88	0.06
N=11	0.262	1.0	1.0	0.115	2.28	0.57	0.063	4.16	0.26	0.047	5.57	0.09
N=12	1.501	1.0	1.0	0.452	3.32	0.83	0.187	8.03	0.5	0.207	7.25	0.11
N=13	8.347	1.0	1.0	2.716	3.07	0.77	0.987	8.46	0.53	1.036	8.06	0.13
N=14	54.966	1.0	1.0	16.517	3.33	0.83	5.948	9.24	0.58	4.964	11.07	0.17

TABLE VI: Parallelization Performance of Backtracking on Find All Solutions

Threads	our approach		referenced approach	
Size/thread	threads	time	threads	time
N=9	9	0.018s	9	0.018s
N=10	10	0.022s	10	0.064s
N=11	11	0.058s	11	0.141s
N=12	12	0.185s	12	0.809s
N=13	13	1.021s	13	5.222s
N=14	14	6.175s	14	35.156s
N=15	15	40.630s	15	4min 32.056s

TABLE VII: Parallelization Performance Comparation Between of Our Approach And Referenced Approach

a larger number of threads does not significantly improve performance on larger problem sizes due to the algorithm's rapidly increasing computational complexity compared to the backtracking algorithm. However, the design of the backtracking algorithm inherently limits its efficiency.

Our parallelized backtracking approach is distinct from the referenced approach[2] in two major aspects. First, the reference approach balances the workload by the first row of the chess board, which means it can only use at most N threads for parallelization. Our approach separate the work to N^2-3N+2 sub-tasks, which can separate the work to more than N threads with a balanced load. Second, the referenced approach uses std::async and our approach uses OpenMP, the overheads of OpenMP is smaller than std::async.

E. Comparison Among Different Parallel Versions

From our experiments, it can be obviously seen that parallel GA works much better than Backtracking when N is large and the requirement is to find a few unique solutions ($N = 60$ on GA is still faster than Backtracking). This phenomenon can primarily be attributed to two factors: (1) GA operate with a population of candidate solutions, enabling parallel processing and simultaneous evaluation of multiple solutions. This renders GA more amenable to parallelization, particularly when the population size and N become larger. (2) The stochasticity and heuristic components introduced during the GA fitting process allow for a more extensive exploration of the solution space within a shorter time frame. In contrast, Backtracking adheres to a deterministic search approach, which, although more thorough, is considerably more time-consuming.

[2]https://rosettacode.org/wiki/N-queens_problem#C++

VI. Conclusions

A. Take-aways

Through analyzing the experimental outcomes, it becomes evident that the GA constitutes an efficacious approach to solving the N-Queen problem, particularly when both the number of queens (N) and the population size (P) are large. Moreover, GA demonstrates its effectiveness in scenarios where finding multiple solutions is required. Given that GA remains a largely invariant algorithm when applied to disparate problems, the proposed generalized framework for parallelizing GA can be extensively employed across various domains, requiring only minimal adjustments. Consequently, it is anticipated that

analogous performance improvement could be observed when the problem utilizing the parallelized GA scales up its size.

When searching for a small number of unique solutions for problem sizes where $N > 20$, genetic algorithms (GA) outperform both backtracking and brute-force search in terms of running time and efficiency. However, for smaller problem sizes where the goal is to find all solutions, backtracking proves to be the superior algorithm. Intrinsically, backtracking is similar to brute-force search, suggesting that there may be room to optimize both algorithms in order to enhance parallelism and reduce problem size.

B. Challenges Faced

Although the Genetic Algorithm has been extensively implemented in various resources, debugging its implementation in C++ proved to be time-consuming, as it required tracking the correctness of chromosome mutations within each population. This process was rather lengthy. In order to parallelize the code, it was crucial to ensure that no contention occurred, which could easily lead to erroneous mutation results and segmentation faults. Consequently, each parallelization was executed with great caution, and multiple runs were conducted to verify the correctness of the algorithm.

Furthermore, the experiments necessitated a comprehensive exploration of the Genetic Algorithm's hyperparameters, as these parameters can significantly influence the algorithm's performance and the number of generations required for convergence. Thus, an extensive set of experiments was performed, involving various hyperparameter configurations, accompanied by meticulous analyses. This thorough experimental approach also contributed to the overall duration of the project.

Parallelizing recursive backtracking algorithms faces limitations in both parallelism and load balancing. Converting recursive methods to bottom-up loops is a challenging process, and due to the various optimizations applied to backtracking methods, the computational complexity of each branch is not easily predictable. On the other hand, brute-force search is straightforward to implement and parallelize but difficult to optimize in this case. Reducing the total number of configurations would necessitate significantly more code.

REFERENCES

[1] M. Bozikovic, M. Golub, and L. Budin. Solving n-queen problem using global parallel genetic algorithm. In *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, volume 2, pages 104–107 vol.2, 2003.

[2] K. D. Crawford. Solving the n-queens problem using genetic algorithms. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, SAC '92, page 1039–1047, New York, NY, USA, 1992. Association for Computing Machinery.

[3] E. Dijkstra. A short introduction to the art of programming, August 1971.

[4] A. Dubey, V. Ellappan, R. Paul, and V. Chopra. Comparative analysis of backtracking and genetic algorithm in n queen's problem. *International Journal of Pharmacy and Technology*, 8:25618–25623, 12 2016.

[5] R. Eglese. Simulated annealing: A tool for operational research. *European Journal of Operational Research*, 46(3):271–281, 1990.

[6] C. Erbas, S. Sarkeshik, and M. M. Tanik. Different perspectives of the n-queens problem. In *Proceedings of the 1992 ACM Annual Conference on Communications*, CSC '92, page 99–108, New York, NY, USA, 1992. Association for Computing Machinery.

[7] V. Jain and J. Prasad. Solving n-queen problem using genetic algorithm by advance mutation operator. *International Journal of Electrical and Computer Engineering (IJECE)*, 8:4519, 12 2018.

[8] S. Katoch, S. S. Chauhan, and V. Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126, Feb 2021.

[9] I. Martinjak and M. Golub. Comparison of heuristic algorithms for the n-queen problem. pages 759 – 764, 07 2007.