N-Queens Problem

Group 10

Talal Ahmad, Lucas Duncan, Ryan Chan, Aaron Kim, Jack Jarrett, Carter Chan, Nick Simbhunauth, Andre Moodley, Lucas Horta, Brayden Van De Wyckel

Overview

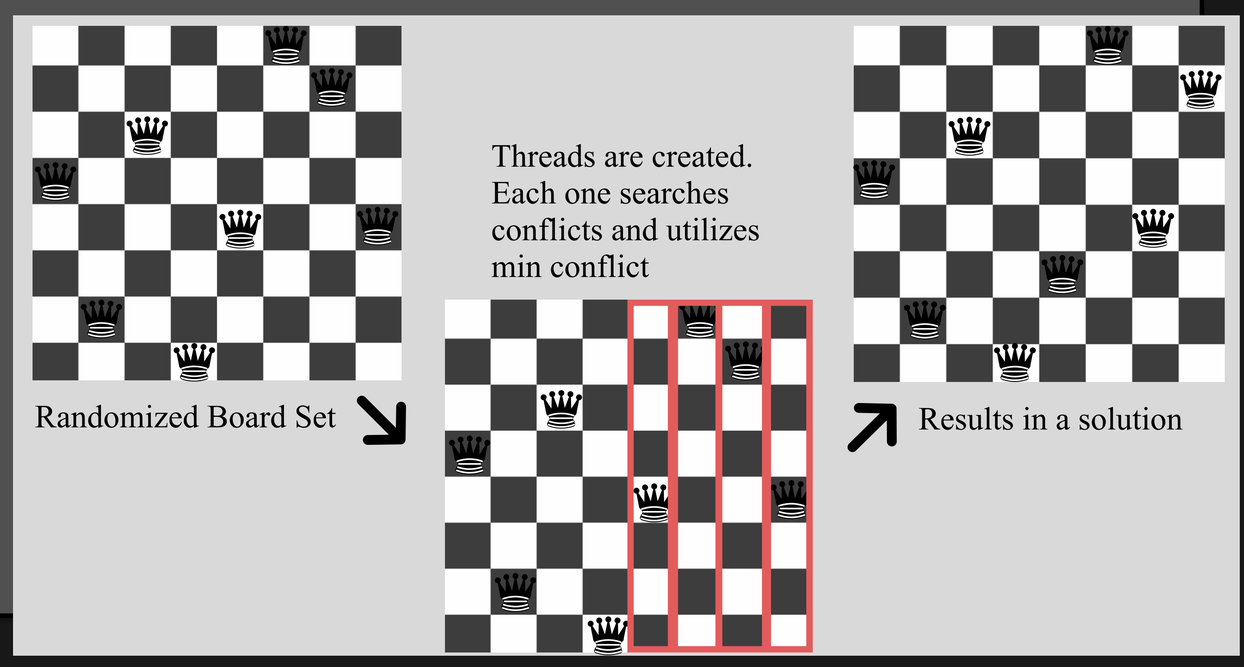
The N-Queens problem involves placing N queens on an N×N chessboard without any two queens attacking each other, meaning no queen cannot share the same row, column, or diagonal.

The goal is to find a valid configuration that satisfies these constraints, or determine that no configuration exists for the given N.

Parallelized Min-Conflict

The Min-Conflicts algorithm is the primary method used to solve the N-Queens problem. It works by iteratively selecting a conflicting queen and moving it to a row that minimizes the number of conflicts. The process repeats until a solution is found or a maximum number of steps is reached.

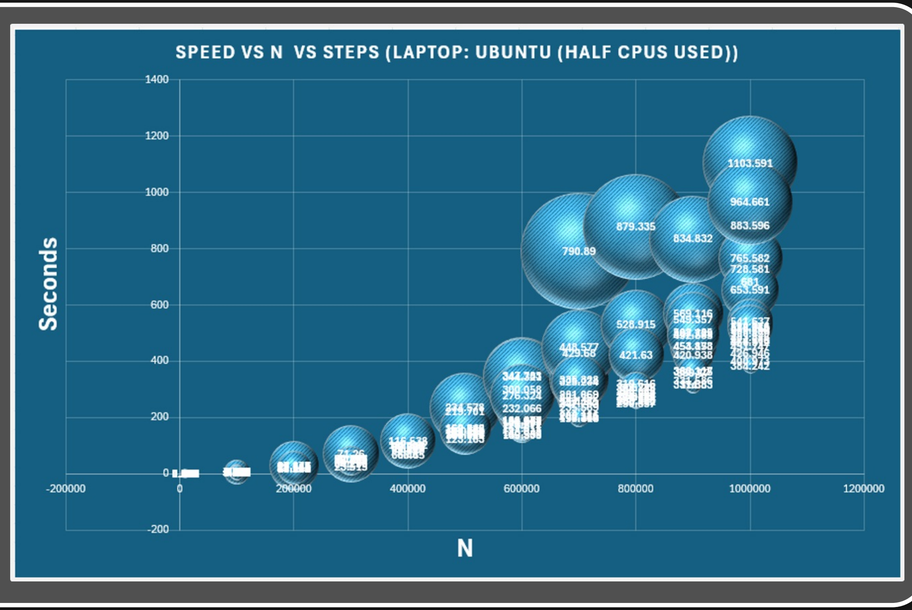
In our implementation, we combined Min-Conflicts with parallelization. Threads independently handled subsets of conflicting columns, adjusting queen positions simultaneously. Atomic operations ensured thread safety while updating shared board state. By merging Min-Conflicts with parallel processing, we drastically improved performance, making it scalable and efficient for solving even very large N-Queens instances.



Runtimes Tests & Results

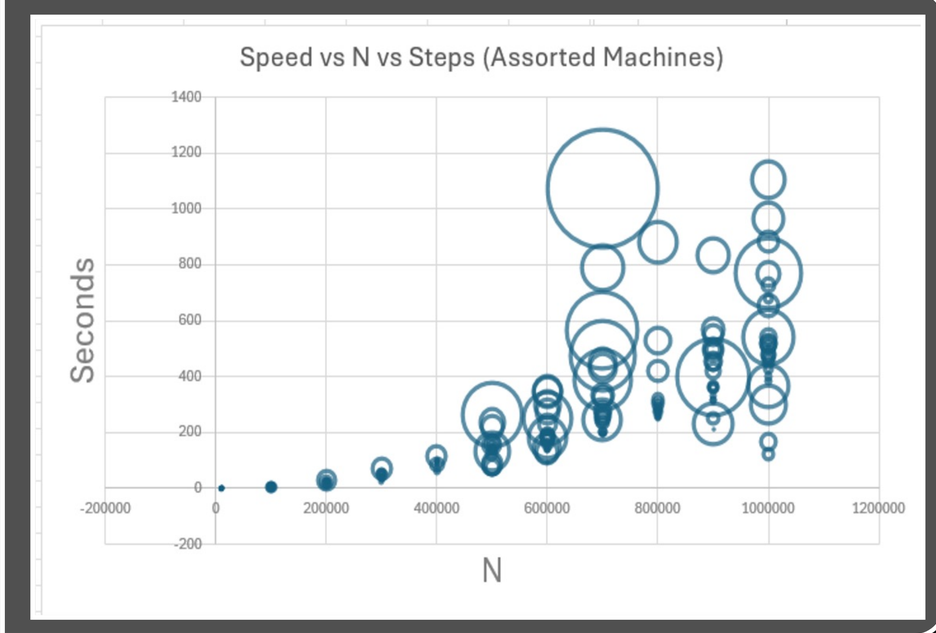
Ubuntu Machine

N	Av Seconds	Av Steps
10000	0.52	112.81
100000	5.46	412.14
200000	22.85	2118.70
300000	46.63	2258.40
400000	90.40	2044.40
500000	157.45	4959.36
600000	212.09	9073.67
700000	311.93	11504.53
800000	345.89	6651.20
900000	455.08	8157.87
1000000	599.03	8656.45



Assorted Machines

N	Av Seconds	Av Steps
10000	0.52	112.81
100000	5.46	412.14
200000	22.85	2118.70
300000	46.63	2258.40
400000	90.40	2044.40
500000	149.91	13106.95
600000	199.09	14478.00
700000	371.26	48219.40
800000	345.89	6651.20
900000	416.49	17356.63
1000000	548.05	18711.85



Design Choices

Algorithm Process

1. Randomized Board Initialization
2. Search for Columns with Conflicts
3. Split conflicted columns evenly between threads
4. Use Min-Conflict Algorithm to place queens in the best row

Optimizations

1. Efficient Random Selection with XORShift:
   - Implemented the memory-efficient XORShift algorithm for random number generation.
2. Use of Atomic Access:
   - Optimized speed and accuracy by balancing atomic accesses and data storage.
3. Thread Pool Implementation:
   - Created a thread pool to reuse threads and avoid the overhead of recreating them.
4. Reusing Memory:
   - Reused allocated memory to reduce overhead and improve performance.

Compilation

```
clang -O3 -march=native -flto -fomit-frame-pointer -o fileoutput new_testable.c
```

The following flags were used in our efficient compilation:

- **-O3**: Enables highest optimizations for speed
- **-march=native**: Tailors the compiled code to the specific CPU architecture in which the compilation takes place
- **-flto**: Applies link-time optimizations to improve overall code efficiency.
- **-fomit-frame-pointer**: Reduces overhead by not maintaining frame pointer in functions, freeing up registers for calculations.

Challenges

Min-Conflict Performance:

- Sequential Min-Conflict was too slow for large N, especially N=1,000,000
- Solution: utilize parallelization in our code to break the problem into subproblems

Python's Inefficiency for large N:

- Solution: changed to Golang and then C for lower level operations