N-Queens Problem
IT301 Parallel Computing Project

Radhika Chhabra
Information Technology(201IT144)
National Institute of Technology
Karnataka Surathkal, India

Dhruvil Lakhtaria
Information Technology(201IT119)
National Institute of Technology
Karnataka Surathkal, India

Rohit Taparia
Information Technology(201IT151)
National Institute of Technology
Karnataka Surathkal, India

***Abstract*—The N-queens problem is a classical combinatorial problem in which it is required to place n queens on an n x n chessboard so no two queens can attack each other, that is so that no two of them are in the same row, column, or diagonal. In this ongoing work we describe a sequential and parallel algorithm for generating solutions of the n-queens problem and its implementation in C, C-OpenMP, C-MPI and C-CUDA. We also compare and conclude which approach is better and why parallel algorithms works better than sequential.**

***Index Terms*—Sequential, OpenMP, MPI, CUDA**

I. Introduction

The n-queens problem is the problem of placing n queens on an n x n chessboard so that no two attack, i.e., so that no two are in the same row, column or diagonal. Thus, solutions to this problem can be represented by n x n permutation matrices, i.e., matrices of zeros and ones in which there are exactly one 1 in every column and every row. For example, Figure 1, where each dot represents a 1 and each blank cell represents a 0, represents a solution to the 5-queens problem.

The number of solutions Q(n) of the n-queens problem grows rapidly for n ³ 6. Table 1 gives several values of Q(n).

6	4
7	40
8	90
9	352
10	754
11	2680
12	14200
13	73712

TABLE I
N VS Q(N)

The idea of the solution is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false. The time complexity for this is O(N!). For large values of N, it can even hang the system. Hence we require parallel processing which can dramatically decrease the time required.The solutions to the n-queens problem can be generated in parallel by using the master-worker technique. The manager generates the upper portion of the search tree by generating those nodes of fixed depth d, for some d. The manager dynamically passes each of these sequences to an idle worker, who in turn continues to search for sequences with n-queens property that contain the fixed subsequence of length d.

A. Parallel Processing

There are (n2!)/(n!(n2n)!) different ways of placing n number of queens on an nxn chessboard but only a small number of them are actual solutions out of which even a smaller number are considered unique solutions. For example, in the case of eight queens problem, there are 4,426,165,368 ways of placing the 8 queens on the board, while only 92 of them are solutions and when you ignore those which are the reflections or rotations of others, only 12 of them remain as the unique solutions. As the result of this significantly large search space, the n-queens problem is computationally expensive and this makes it perfect for the parallel programming paradigm where the main problem is divided into a number of subproblems and sent to a number of processing units, in order to avoid putting a considerable amount of computation load on a single unit and to speed up the process of finding the results.

.

II. Literature Survey

The research paper[1] deals with the problem of N-Queen. It describe a parallel algorithm for generating solutions of the n-queens problem and its implementation in C-MPI. The research paper discuses the backtracking approach and explain its algorithm. The parallel execution is done using master-worker technique. It is implemented using MPI. It concluded that parallel execution speed up the algorithm execution and return the result faster.

.

III. Methodology

A. Sequential Approach

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

Algorithm :

1) Start in the leftmost column.
2) If all queens are placed, return true
3) Try all rows in the current column .Do following for every tried row.
    a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    b) If placing the queen in [row, column] leads to a solution then return true.
    c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step 1) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

Code:

```
bool solveNQ()
{
    int n;
    cin>>n;
    vector<vector<int>>board(n,vector<int>(n));
    if(solveNQUtil(board, 0) == false){
        cout << "Solution does not exist";
        return false;
    }
    return true;
}
```

Util function for solving N-Queens:

```
bool solveNQUtil(vector<vector<int>>&board,
             int col)
{
    int n = board.size();
    if (col >= n)
        return true;
    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}
```

Function to check whether it is safe to place queen or not:

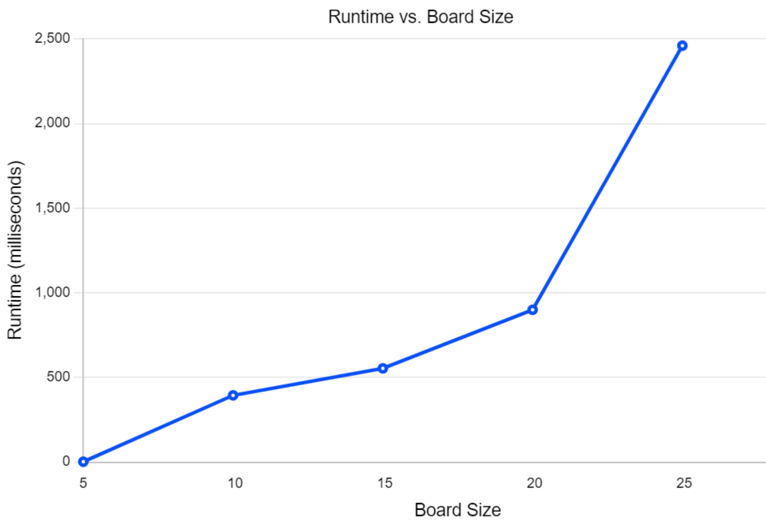
```
bool isSafe(vector<vector<int>>&board,int row,
            int col)
{
    int i, j;
    int n = board.size();
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (i=row,j=col;i>=0&&j>=0;i--,j--)
        if (board[i][j])
            return false;

    for (i=row,j=col;j>=0&&i<n;i++,j--)
        if (board[i][j])
            return false;
    return true;
}
```

IV. Result

We have completed the sequential approach and analysed its results. The run time varies for different board size depending on the size of the recursive tree. We can see the variation of the timing in following table.

Input Size	Runtime
5	2.311ms
10	394.331ms
15	553.71ms
20	899.63ms
25	2459.42ms

TABLE II
N VS Q(N)



.

V. Conclusion

1) We can clearly say that the sequential approach is good only for small sized boards.
2) When we increase the size of the board the run time increases significantly and system also may hang.
3) There is a need to parallelise the main algorithm behind N-Queens problem.
4) We can use OpenMP,MPI and CUDA to do the task and verify for the best runtime.

.

References

[1] Kesri, Vishal Kesri, Vaibhav Pattnaik, P.K.. (2012). An Unique Solution for N queen Problem. International Journal of Computer Applications. 43. 1-6.

[2] A. Ayala et al., ”Accelerating N-queens problem using OpenMP,” 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2011, pp. 535-539.

[3] A. Khademzadeh, M. A. Sharbaf and A. Bayati, ”An Optimized MPI-based Approach for Solving the N-Queens Problem,” 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2012, pp. 119-124.

[4] Feinbube, Frank Rabe, Bernhard Löwis, Martin Polze, Andreas. (2010). NQueens on CUDA: Optimization Issues. 63-70.