Problem Statment : The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. The worst case “brute force” solution for the N-queens puzzle has an O(n^n),Backtracking is a recursive method which starts a queen at an edge and, ideally, saves the possible attack positions.It has complexity of O(2^n).We can use the parallelism to solve this problem and print all the solutions for given N.

H/W and S/W Requirement : Python,Jupyter Notebook,Libraries like numpy,pandas,sklearn etc.,64 bit OS, 8 GB RAM, 500 GB HDD,Monitor,Keyboard.

Abstract :

1.The ultimate goal of the N-Queens problem is to find the total number of distinct solutions to place a given number of chess queens on a quadratic chess board with a specific number of squares on one side of a board (N) in a way that no two queens would be able to attack each other.

2.Two queens can attack each other when they are placed on the same horizontal row, vertical column or in one of (2*(2*n - 1)) possible diagonals.

3.For example, on an eight-by-eight board, there are 92 distinct solutions. The difference between a unique and distinct solution is that distinct solutions allow for symmetrical operations, like rotations and reflections of the board, to be counted as one; so an eight-by-eight board has 12 unique solutions.

4.The most common approach is to use the brute-force approach, which is to simply test each position for each board size.

5.Unfortunately, trying a brute-force algorithm by testing if every position on the board is valid will inevitably lead to a huge number of positions to validate (n 2)!/((n 2)-n)!; even for a four-by-four board the possible number of positions is 16!/12! = 43680.

6.Applying a heuristic approach by only allowing one queen per row reduces the problem size to a vector of n elements, where each element holds the position in the row and the number of possible solutions reduces to nn.

7.OpenMP is an industry standard for portable multi-threaded application development. This approach is effective at fine-grain (loop-level) and large-grain (function-level) threading.

Objective : To understand usage of parallel algorithms to solve N Queen’s problem with lesser time complexity

Scope :

1.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.

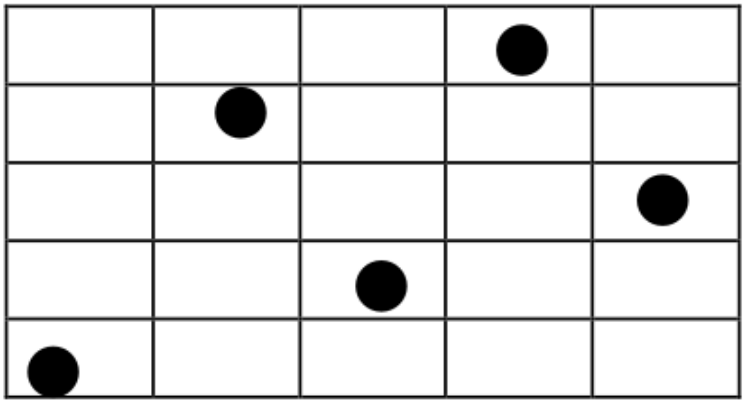


Figure 1. Solution to 5-queens problem

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

Table 1. Values of *Q(n)*

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

3.The usual method for finding solutions to the n-queens problem uses “backtracking,” a general procedure for solving a problem by systematically generating all possible solutions.

4.By using parallel implementation we can decrease the execution time.

System Architecture:

1.System basically has used c++,open mp and google colab for the implementation purpose.

2.Backtracking can be described by a search tree in which each node corresponds to a partial solution. Going down the tree corresponds to progress toward obtaining a complete solution.

3.Going up the tree, i.e.,backtracking corresponds to returning to a partial solution from which it might be hopeful to proceed forward again.

4.OpenMP directives provide an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multi-core and symmetric multiprocessor systems.

```
void solve() {

#pragma omp parallel for

for(int i=0; i<size; i++) {

// try all positions in first row

// create separate array for each recursion

// started here

setQueen(new int[size], 0, i);

}

}
```

5.The important function in our program is the solve function. Solve is easy to parallelize as the solutions are independent (review the search tree). The sample uses the OpenMP #pragma parallel for to parallelize the important loop.

Test Cases -

N Value	Actual Output	Expected Output	Test Case
8	92	92	Pass
10	754	754	Pass
12	14200	14200	Pass

Results :

Parallel Code :

Output:92

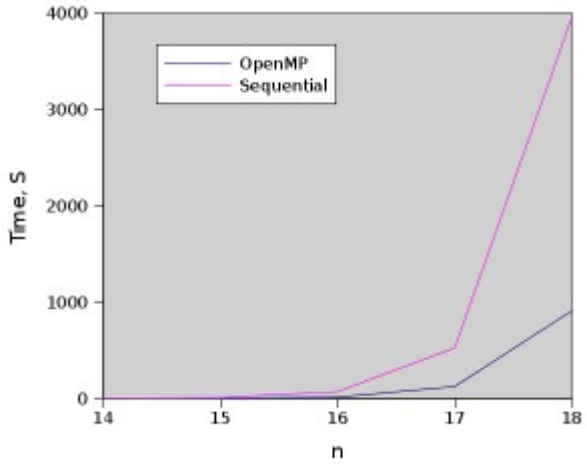
Time taken for execusion is :0.00142119

Serial Code:

92

0.00214683

Conclusion:



From this graph, OpenMP seems to be doing a much better job than just the sequential implementation.

Parallel Algorithm has lesser exceution time than the serial one.

References :

1.Lorna E. Salaman Jorge “A Parallel Algorithm for the n-Queens Problem”.

2.www.geeksforgeeks.com

3.Parallelizing N-Queens with Intel® Parallel Composer

Parallel code:

```
%%writefile testp.cpp
#include <bits/stdc++.h>
#include <omp.h>

using namespace std;

long long int nrOfSolutions=0;
int size=0;
int nrOdd = 0;
long long int arr[500];
int LEVEL = 3;
int cnt[20] ={0};

void setQueen(int queens[], int row, int col, int id, int col1) {
for(int i=0; i<row; i++) {
// vertical attacks
if (queens[i]==col) {
return;
}
// diagonal attacks
if (abs(queens[i]-col) == (row-i) ) {
return;
}
}

queens[row]=col;
if(row==size-1 && col1!=(size/2)) {
arr[omp_get_thread_num()]++;
}
else if(row==size-1 && col1==(size/2)){
nrOdd++;
}
else {
for(int i=0; i<size; i++) {
setQueen(queens, row+1, i, id,col1);
}
}
}

void setQueen1(int queens[], int row, int col, int id, int col1) {
for(int i=0; i<row; i++) {
// vertical attacks
if (queens[i]==col) {
return;
}
// diagonal attacks
if (abs(queens[i]-col) == (row-i) ) {
return;
```

```
}
}

queens[row]=col;
if(row==size-1 && col1!=(size/2)) {
arr[omp_get_thread_num()]++;
}
else if(row==size-1 && col1==(size/2)){
nrOdd++;
}
else {
if(row<=LEVEL){
for(int i=0; i<size; i++) {
#pragma omp parallel
#pragma omp single
{
#pragma omp task
setQueen1(queens, row+1, i, id,col1);
}
}
}
else{
for(int i=0; i<size; i++) {
setQueen1(queens, row+1, i, id,col1);
}
}
}
}

void solve() {
int myid=0 ;

if(size%2==0){
#pragma omp parallel
#pragma omp single
{
for(int i=0; i<(size/2)-3; i++) {
#pragma omp task
setQueen(new int[size], 0, i, myid,i);
}
for(int i=(size/2)-3;i<(size/2);i++){
setQueen1(new int[size], 0, i, myid,i);
}
}

}
else{
#pragma omp parallel
#pragma omp single
{
for(int i=0; i<=(size/2); i++) {
```

```
#pragma omp task
setQueen(new int[size], 0, i, myid,i);
}
}
}

}

int main(int argc, char*argv[]) {

int queens = 8;
size = queens;
for(int i=0;i<500;i++){
arr[i] = 0;
}
double sTime = omp_get_wtime();
solve();
double eTime = omp_get_wtime();
long long int ans = 0;
int mn = 1000;
for(int i=0;i<500;i++){
if(arr[i]==0){
mn = min(mn,i);
}
nrOfSolutions += arr[i];
}
if(size%2==0){
ans = nrOfSolutions*2;
}
else{
ans = (nrOfSolutions*2)+nrOdd;
}
cout<<ans<<endl;
cout<<"Time taken for execusion is :"<<(eTime-sTime)<<endl;
return 0;
}


%%script bash
g++ -fopenmp -o nqu testp.cpp
ls -laX
./nqu
```

Output:92

```
Time taken for execusion is :0.00142119
```

Sequential Code:

```
%%writefile seq.cpp
```

```
#include<chrono>
#include <omp.h>
#include<bits/stdc++.h>
#define MAX_N 8
using namespace std;
using namespace std::chrono;
bool isSafe(vector<vector<string> >& board, int row, int col, int N)
{
//check left side rows
for (auto j = 0; j<col; ++j)
if (board[row][j] == "Q")
return false;
//check top left side diagonal
for (auto i = row, j = col; i>-1 && j>-1; --i, --j)
if (board[i][j] == "Q")
return false;
//check bottom left side diagonal
for (auto i = row, j = col; i<N && j>-1; ++i, --j)
if (board[i][j] == "Q")
return false;
return true;
}
bool backtracking(vector<vector<string> >& board, int col, vector<vector<string> >& testboard, int N)
{
if (col == N)
{
vector<string> emptyRow;
board.emplace_back(emptyRow);
int size = board.size();
for (auto i = 0; i<N; ++i)
{
string row = "";
for (auto j = 0; j<N; ++j)
row += testboard[i][j];
board[size-1].emplace_back(row);
}
return false;
}
for (auto i = 0; i<N; ++i)
{
if (isSafe(testboard, i, col, N))
{
testboard[i][col] = "Q";
if (!backtracking(board, col+1, testboard, N))
testboard[i][col] = ".";
else
return true;
}
}
return false;
}
```

```
vector<vector<string> > solveNQueens(int A) {
vector<vector<string> > testboard;
if (A==2 || A==3)
return testboard;
vector<string> row(A, ".");
for (auto i = 0; i<A; ++i)
testboard.emplace_back(row);
vector<vector<string> > board;
backtracking(board, 0, testboard, A);
return board;
}

int main(){
int cnt=0;
//auto start = high_resolution_clock::now();
double sTime = omp_get_wtime();
vector<vector<string>> ans = solveNQueens(8);
double eTime = omp_get_wtime();
for( auto const& string_vec : ans )
{
for( auto const& s : string_vec )
{
cout << s << endl;
}
cnt++;
cout<<"==============\n";
}
// auto stop = high_resolution_clock::now();
cout<<"Time taken for execusion is :"<<(eTime-sTime)<<endl;
// Get duration. Substart timepoints to
// get durarion. To cast it to proper unit
// use duration cast method
// auto duration = duration_cast<milliseconds>(stop - start);
//cout << "Time taken by function: "
//<< duration.count() << " milliseconds" << endl;
cout<<cnt;
}


%%script bash

g++ -fopenmp -o myseq seq.cpp
ls -laX
./myseq
```

Output :Time taken for execusion is : 0.00214683

92