Distributed Solution to the K-Queens Problem Using MPI

Antonio Ruiz
Dr.Wolffe
Project 4 CIS 677

December 8, 2018

Abstract

The purpose of the assignment is to write a program that solves the infamous K-queens problem. The problem involves finding the number of possible placement of K queens on a K by K chessboard. The number of possible placements exponentially increases as K increases. We are to write a parallelized program using MPI to distribute the intensive work amongst a virtual cluster.

Serial Compilation

- `gcc kqueens.c -lm -o kqueens`
- `./kqueens <k>`

MPI Compilation

- `mpicc mpi_queens.c -lm -o mpi_queens`
- `mpirun -np <nodes> --hostfile my_hosts ./mpi_queens <k>`

PARAMETERS

- k: The number of dimensions (k by k) and queens
- nodes: The number of nodes in virtual cluster

Algorithm: Serialized Version

The serialized version is an implementation of Wirth's Algorithm. We simplify our approach by using a solution vector to give hold the placements of Queens on the board. This means that only one Queen can exist in one column at any time. We can think of the starting position as being the top-left corner of the chessboard. This is a recursive solution where we start a position in column 1 and examine all possible solutions starting from that position. We use backtracking to get out of paths that lead to conflicts between Queens.

Algorithm 1 Serialized Version

```
procedure PlaceQueen(int k , int column, int * solution)
    if k == col then
        Solution++
    else
        for j = 1..k do
            if checkrows()==0 and checkdiagonals()==0 then
                solution[col] ← j
                Place_Queen(k, column + 1, solution)
```

While this is an elegant solution, it does not scale well. Our recursive tree grows sporadically as the number possible solutions increases with K.

Algorithm: Parallelized Version

We use the same PlaceQueens algorithm described in our serialized solution, but we partition the work amongs nodes on a Virtual cluster. As stated above, this is a K by K chessboard and we are examining the number of possible placements of K Queens starting from column 1. Column 1 has K possible starting solutions, so we divide those K possible starting positions among K nodes in a cluster. A master node will receive messages as to how many possible messages are received from each worker. This leads to a limitation for the program, for a K by K chessboard, we need K + 1 Nodes to be present in our cluster.

Algorithm 2 Parallelize Version

if *myrank != MASTER* **then**
  $solution[0] \leftarrow myrank$
  $Place_{Q}ueen(k, 1, myrank, solution)$
else
  for `source = 1..NumberOfNodes` **do**
    recieve the responses from all worker nodes
    $SOLUTIONS \leftarrow$ *sum of all responses*

We partition the work by distributing among K nodes. Each node will start their own recursive tree and find their individual number of solutions and send it to the Master node. The Master node will then go through each worker and receive their number of solutions through a response variable. The sum of all these variables will be stored as SOLUTIONS.

Accuracy of Models

In this section we analyzed how accurate were the models to one another. We can see that both approaches return the same number of solutions.

```
Solution matrix
Number of solutions for both approaches.
K = 8,10,12,14,16

           8     10     12        14       16
-----------------------------------------------------------
sequential  | 92 | 724 |  14200 | 365596 |   14772512 |
-----------------------------------------------------------
mpi         | 92 | 724 |  14200 | 365596 |   14772512 |
-----------------------------------------------------------

RESULTS: SEQUENTIAL K = 8

[1 5 8 6 3 7 2 4 ]
[1 6 8 3 7 4 2 5 ]
[1 7 4 6 8 2 5 3 ]
[1 7 5 8 2 4 6 3 ]
[2 4 6 8 3 1 7 5 ]
[2 5 7 1 3 8 6 4 ]
[2 5 7 4 1 8 6 3 ]
[2 6 1 7 4 8 3 5 ]
[2 6 8 3 1 4 7 5 ]
[2 7 3 6 8 5 1 4 ]
[2 7 5 8 1 4 6 3 ]
[2 8 6 1 3 5 7 4 ]
[3 1 7 5 8 2 4 6 ]
[3 5 2 8 1 7 4 6 ]
[3 5 2 8 6 4 7 1 ]
[3 5 7 1 4 2 8 6 ]
[3 5 8 4 1 7 2 6 ]
[3 6 2 5 8 1 7 4 ]
[3 6 2 7 1 4 8 5 ]
[3 6 2 7 5 1 8 4 ]
[3 6 4 1 8 5 7 2 ]
```

```
[3 6 4 2 8 5 7 1 ]
[3 6 8 1 4 7 5 2 ]
[3 6 8 1 5 7 2 4 ]
[3 6 8 2 4 1 7 5 ]
[3 7 2 8 5 1 4 6 ]
[3 7 2 8 6 4 1 5 ]
[3 8 4 7 1 6 2 5 ]
[4 1 5 8 2 7 3 6 ]
[4 1 5 8 6 3 7 2 ]
[4 2 5 8 6 1 3 7 ]
[4 2 7 3 6 8 1 5 ]
[4 2 7 3 6 8 5 1 ]
[4 2 7 5 1 8 6 3 ]
[4 2 8 5 7 1 3 6 ]
[4 2 8 6 1 3 5 7 ]
[4 6 1 5 2 8 3 7 ]
[4 6 8 2 7 1 3 5 ]
[4 6 8 3 1 7 5 2 ]
[4 7 1 8 5 2 6 3 ]
[4 7 3 8 2 5 1 6 ]
[4 7 5 2 6 1 3 8 ]
[4 7 5 3 1 6 8 2 ]
[4 8 1 3 6 2 7 5 ]
[4 8 1 5 7 2 6 3 ]
[4 8 5 3 1 7 2 6 ]
[5 1 4 6 8 2 7 3 ]
[5 1 8 4 2 7 3 6 ]
[5 1 8 6 3 7 2 4 ]
[5 2 4 6 8 3 1 7 ]
[5 2 4 7 3 8 6 1 ]
[5 2 6 1 7 4 8 3 ]
[5 2 8 1 4 7 3 6 ]
[5 3 1 6 8 2 4 7 ]
[5 3 1 7 2 8 6 4 ]
[5 3 8 4 7 1 6 2 ]
[5 7 1 3 8 6 4 2 ]
[5 7 1 4 2 8 6 3 ]
[5 7 2 4 8 1 3 6 ]
[5 7 2 6 3 1 4 8 ]
[5 7 2 6 3 1 8 4 ]
```

```
[5 7 4 1 3 8 6 2 ]
[5 8 4 1 3 6 2 7 ]
[5 8 4 1 7 2 6 3 ]
[6 1 5 2 8 3 7 4 ]
[6 2 7 1 3 5 8 4 ]
[6 2 7 1 4 8 5 3 ]
[6 3 1 7 5 8 2 4 ]
[6 3 1 8 4 2 7 5 ]
[6 3 1 8 5 2 4 7 ]
[6 3 5 7 1 4 2 8 ]
[6 3 5 8 1 4 2 7 ]
[6 3 7 2 4 8 1 5 ]
[6 3 7 2 8 5 1 4 ]
[6 3 7 4 1 8 2 5 ]
[6 4 1 5 8 2 7 3 ]
[6 4 2 8 5 7 1 3 ]
[6 4 7 1 3 5 2 8 ]
[6 4 7 1 8 2 5 3 ]
[6 8 2 4 1 7 5 3 ]
[7 1 3 8 6 4 2 5 ]
[7 2 4 1 8 5 3 6 ]
[7 2 6 3 1 4 8 5 ]
[7 3 1 6 8 5 2 4 ]
[7 3 8 2 5 1 6 4 ]
[7 4 2 5 8 1 3 6 ]
[7 4 2 8 6 1 3 5 ]
[7 5 3 1 6 8 2 4 ]
[8 2 4 1 7 5 3 6 ]
[8 2 5 3 1 7 4 6 ]
[8 3 1 6 2 5 7 4 ]
[8 4 1 3 6 2 7 5 ]
Found 92


RESULTS: MPI , K = 8


PROCESS 1 [1 5 8 6 3 7 2 4 ]
PROCESS 1 [1 6 8 3 7 4 2 5 ]
PROCESS 1 [1 7 4 6 8 2 5 3 ]
```

```
PROCESS 1 [1 7 5 8 2 4 6 3 ]
Process 1 computed 4 solutions
--------------------------------
PROCESS 6 [6 1 5 2 8 3 7 4 ]
PROCESS 7 [7 1 3 8 6 4 2 5 ]
PROCESS 6 [6 2 7 1 3 5 8 4 ]
PROCESS 6 [6 2 7 1 4 8 5 3 ]
PROCESS 7 [7 2 4 1 8 5 3 6 ]
PROCESS 7 [7 2 6 3 1 4 8 5 ]
PROCESS 7 [7 3 1 6 8 5 2 4 ]
PROCESS 7 [7 3 8 2 5 1 6 4 ]
PROCESS 7 [7 4 2 5 8 1 3 6 ]
PROCESS 7 [7 4 2 8 6 1 3 5 ]
PROCESS 7 [7 5 3 1 6 8 2 4 ]
PROCESS 6 [6 3 1 7 5 8 2 4 ]
PROCESS 6 [6 3 1 8 4 2 7 5 ]
PROCESS 6 [6 3 1 8 5 2 4 7 ]
PROCESS 6 [6 3 5 7 1 4 2 8 ]
PROCESS 6 [6 3 5 8 1 4 2 7 ]
PROCESS 6 [6 3 7 2 4 8 1 5 ]
PROCESS 6 [6 3 7 2 8 5 1 4 ]
PROCESS 6 [6 3 7 4 1 8 2 5 ]
PROCESS 6 [6 4 1 5 8 2 7 3 ]
PROCESS 6 [6 4 2 8 5 7 1 3 ]
PROCESS 6 [6 4 7 1 3 5 2 8 ]
PROCESS 6 [6 4 7 1 8 2 5 3 ]
PROCESS 6 [6 8 2 4 1 7 5 3 ]
PROCESS 8 [8 2 4 1 7 5 3 6 ]
PROCESS 8 [8 2 5 3 1 7 4 6 ]
PROCESS 8 [8 3 1 6 2 5 7 4 ]
PROCESS 8 [8 4 1 3 6 2 7 5 ]
PROCESS 2 [2 4 6 8 3 1 7 5 ]
PROCESS 2 [2 5 7 1 3 8 6 4 ]
PROCESS 2 [2 5 7 4 1 8 6 3 ]
PROCESS 4 [4 1 5 8 2 7 3 6 ]
PROCESS 4 [4 1 5 8 6 3 7 2 ]
PROCESS 4 [4 2 5 8 6 1 3 7 ]
PROCESS 4 [4 2 7 3 6 8 1 5 ]
PROCESS 4 [4 2 7 3 6 8 5 1 ]
PROCESS 4 [4 2 7 5 1 8 6 3 ]
```

```
PROCESS 4 [4 2 8 5 7 1 3 6 ]
PROCESS 4 [4 2 8 6 1 3 5 7 ]
PROCESS 4 [4 6 1 5 2 8 3 7 ]
PROCESS 4 [4 6 8 2 7 1 3 5 ]
PROCESS 3 [3 1 7 5 8 2 4 6 ]
PROCESS 3 [3 5 2 8 1 7 4 6 ]
PROCESS 3 [3 5 2 8 6 4 7 1 ]
PROCESS 3 [3 5 7 1 4 2 8 6 ]
PROCESS 3 [3 5 8 4 1 7 2 6 ]
PROCESS 3 [3 6 2 5 8 1 7 4 ]
PROCESS 3 [3 6 2 7 1 4 8 5 ]
PROCESS 3 [3 6 2 7 5 1 8 4 ]
PROCESS 3 [3 6 4 1 8 5 7 2 ]
PROCESS 3 [3 6 4 2 8 5 7 1 ]
PROCESS 3 [3 6 8 1 4 7 5 2 ]
PROCESS 3 [3 6 8 1 5 7 2 4 ]
PROCESS 3 [3 6 8 2 4 1 7 5 ]
PROCESS 4 [4 6 8 3 1 7 5 2 ]
PROCESS 4 [4 7 1 8 5 2 6 3 ]
PROCESS 4 [4 7 3 8 2 5 1 6 ]
PROCESS 4 [4 7 5 2 6 1 3 8 ]
PROCESS 4 [4 7 5 3 1 6 8 2 ]
PROCESS 4 [4 8 1 3 6 2 7 5 ]
PROCESS 4 [4 8 1 5 7 2 6 3 ]
PROCESS 4 [4 8 5 3 1 7 2 6 ]
PROCESS 2 [2 6 1 7 4 8 3 5 ]
PROCESS 2 [2 6 8 3 1 4 7 5 ]
PROCESS 2 [2 7 3 6 8 5 1 4 ]
PROCESS 2 [2 7 5 8 1 4 6 3 ]
PROCESS 2 [2 8 6 1 3 5 7 4 ]
PROCESS 3 [3 7 2 8 5 1 4 6 ]
PROCESS 3 [3 7 2 8 6 4 1 5 ]
PROCESS 3 [3 8 4 7 1 6 2 5 ]
PROCESS 5 [5 1 4 6 8 2 7 3 ]
PROCESS 5 [5 1 8 4 2 7 3 6 ]
PROCESS 5 [5 1 8 6 3 7 2 4 ]
PROCESS 5 [5 2 4 6 8 3 1 7 ]
PROCESS 5 [5 2 4 7 3 8 6 1 ]
PROCESS 5 [5 2 6 1 7 4 8 3 ]
PROCESS 5 [5 2 8 1 4 7 3 6 ]
```

```
PROCESS 5 [5 3 1 6 8 2 4 7 ]
PROCESS 5 [5 3 1 7 2 8 6 4 ]
PROCESS 5 [5 3 8 4 7 1 6 2 ]
PROCESS 5 [5 7 1 3 8 6 4 2 ]
PROCESS 5 [5 7 1 4 2 8 6 3 ]
PROCESS 5 [5 7 2 4 8 1 3 6 ]
PROCESS 5 [5 7 2 6 3 1 4 8 ]
PROCESS 5 [5 7 2 6 3 1 8 4 ]
PROCESS 5 [5 7 4 1 3 8 6 2 ]
PROCESS 5 [5 8 4 1 3 6 2 7 ]
PROCESS 5 [5 8 4 1 7 2 6 3 ]
Process 2 computed 16 solutions
--------------------------------
Process 3 computed 16 solutions
--------------------------------
Process 4 computed 8 solutions
--------------------------------
Process 5 computed 4 solutions
--------------------------------
Process 6 computed 8 solutions
--------------------------------
Process 7 computed 18 solutions
--------------------------------
Process 8 computed 18 solutions
--------------------------------
Total Number of Solutions 92
```

Visualization

Figure 1:

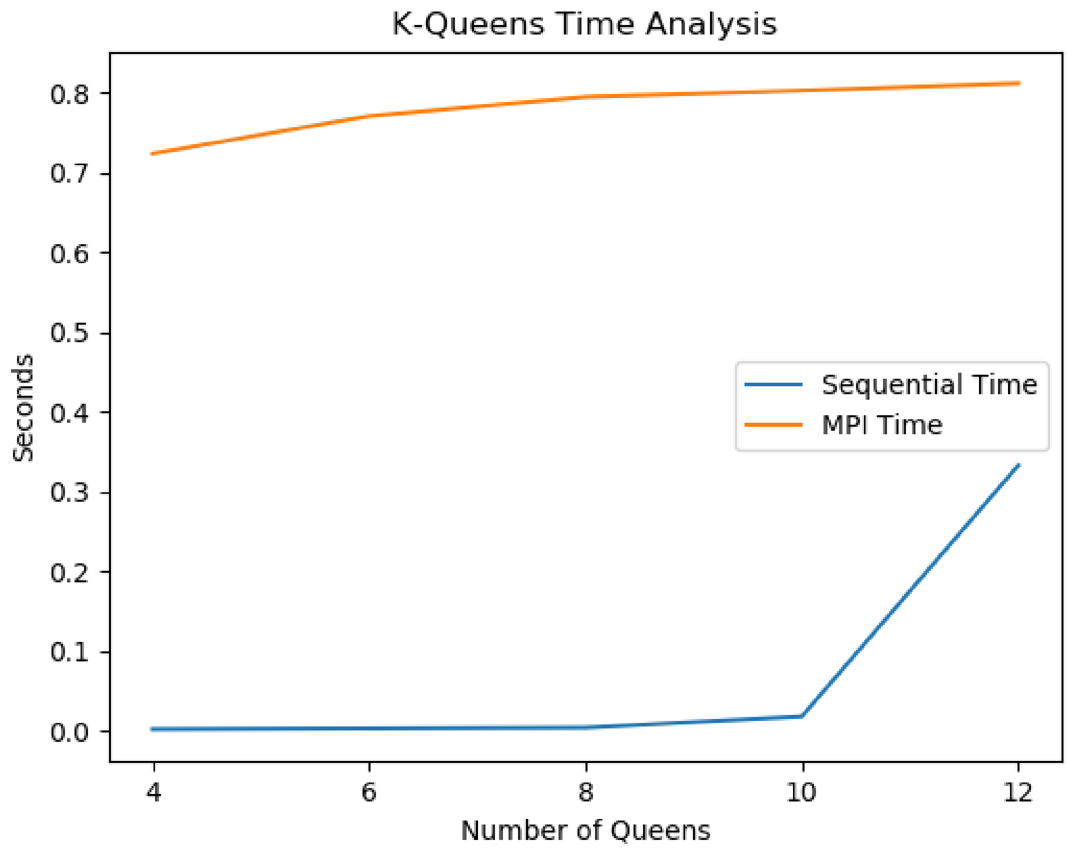
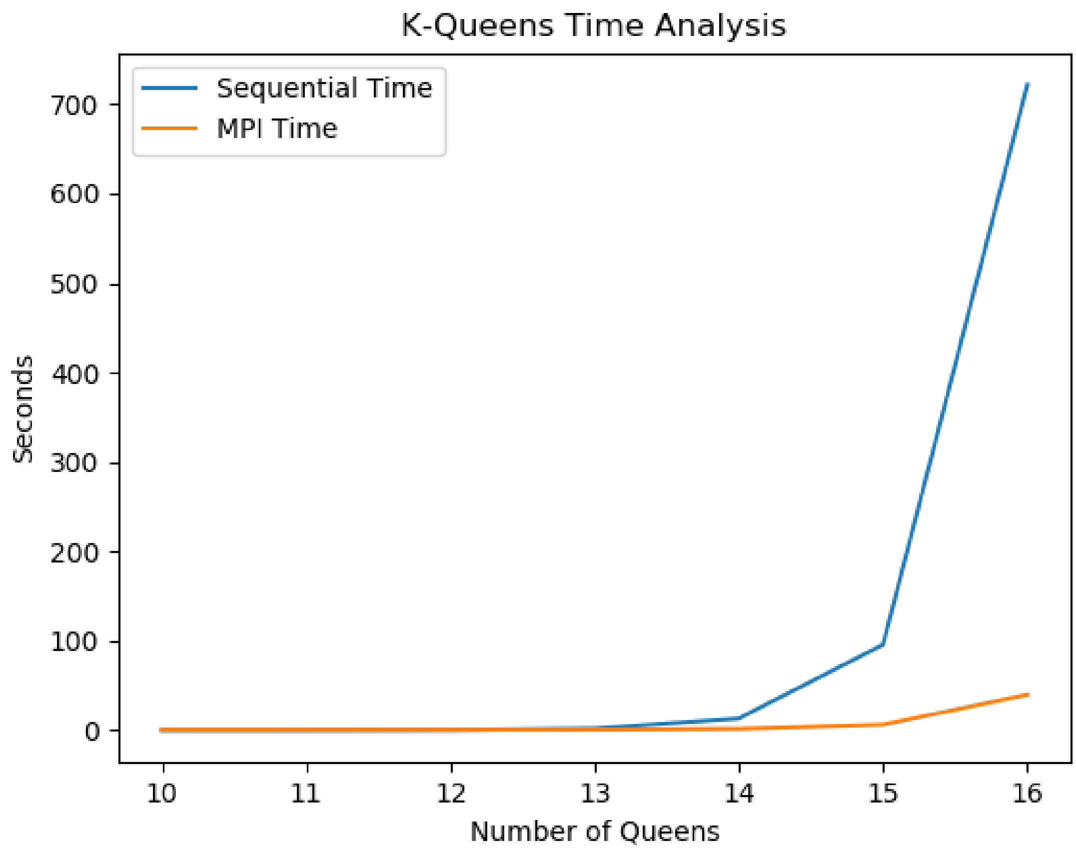


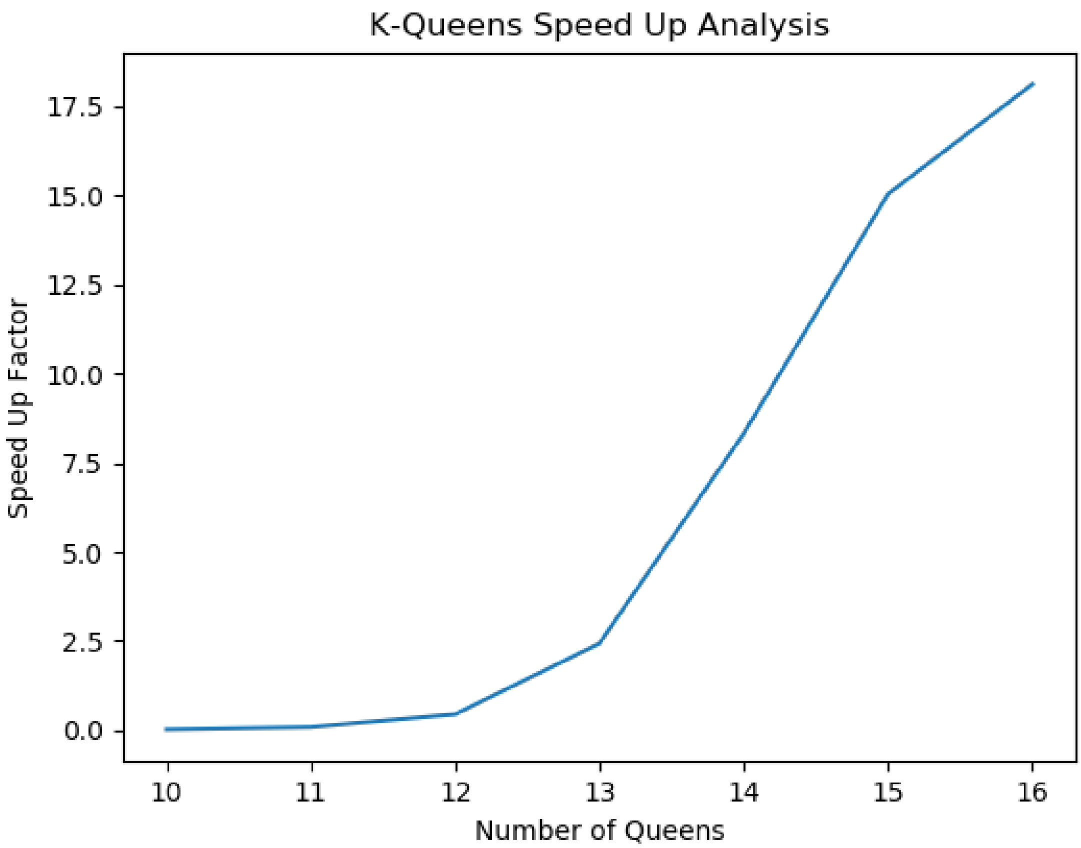
Figure 2:


K-Queens Time Analysis
Sequential Time
MPI Time
700
600
500
400
300
200
100
0
Seconds
10
11
12
13
14
15
16
Number of Queens

Time Complexity Analysis

Our parallel program suffers from the overhead communication for small k values. For large k-values we see that our parallelized version does not suffer from such a drastic exponential growth. However, it does still grow exponentially. This means that the payoff for partitioning the work will still eventually suffer from a bottleneck. Our parallel version is ideal for k-values 14-18, but our speedup after that begins to wane off. The work of each individual worker is still growing exponentially. In a sense, our parallel version is essentially prolonging the inevitable growth curve. We also limit our parallelism to K, which is not ideal if we wanted to mitigate that exponential growth by adding more and more nodes to high values of k.

Figure 3:


K-Queens Speed Up Analysis
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Speed Up Factor
10
11
12
13
14
15
16
Number of Queens

CODE: Serialized Version

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>


void place_queen(int,int, int []);
int check_rows(int, int, int[]);
int check_diagonals(int, int, int []);
void initialize_array(int, int []);
void printArray(int ,int []);

int Solutions;

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


   int k = 0;
   if (argc != 2) {
   fprintf (stderr, "need a k for K Queens Problem!!!!\n");
   exit(-1);
   }

   k = atoi (argv[1]);

   int solution[k];
   //int start = 0;


   initialize_array(k, solution);

   place_queen(k, 0, solution);
   printf("Found %d\n", Solutions);
}
```

```c
void place_queen(int k, int col, int * solution)
{

   if (k == col)
   {
      printArray(k,solution);
      Solutions++;

   }

   for(int row=1; row <= k; row++)
   {
   // check if queen can be placed safely
   if(check_rows(row, col,solution) == 0 && check_diagonals(row,
       col,  solution) == 0)
   {
      //place queen on this row
      solution[col] = row;
      // try rest of problem
      place_queen (k,col+1, solution);

   }//end inner if

   }//end loop
}//end method

//will take a row and column and see if a queen is in danger
int check_rows(int row, int col, int * solution){
   //queens cant exist on the same row
   for(int i = 0; i < col; i ++){
      if(solution[i] == row ){
         return -1;
      }
   }
   return 0;
}


int check_diagonals(int row, int col, int * solution){
   //values we will be examining
```

```
    int danger1;
    int danger2;


    for(int i = 0; i < col; i ++){
        danger1 = solution[i]+i; // diagonals from left side
        danger2 = i - solution[i]; //diagonals from rights side

        if(row + col == danger1){ //left hand diagonals
            return -1;
        }
        if( col - row == danger2){ //right hand diagonals
            return -1;
        }
    }
    return 0;

}

//fill array with zeros
void initialize_array(int k, int * array){
    for (int i = 0; i < k; i++){
        array[i] = 0;
    }
}

//print array
void printArray(int k, int * array){
    printf("[");
    for (int i = 0; i < k; i++){
        printf("%d ", array[i]);
    }
    printf("]\n");

}
```

CODE: Parallelized Version

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>
#include <time.h>
#include <unistd.h>

void place_queen(int,int,int , int []);
int check_rows(int, int, int[]);
int check_diagonals(int, int, int []);
void initialize_array(int, int []);
void printArray(int,int ,int []);

int Solutions;

#define MASTER 0
#define TAG    0

int main (int argc, char *argv[]) {
   int k = 0;
   if (argc != 2) {
      fprintf (stderr, "need a k for K Queens Problem!!!!\n");
      exit(-1);
   }
   int my_rank, num_nodes, source;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);
   k = atoi (argv[1]);


   int solution[k];
   //int start = 0;



   if (my_rank != MASTER) {
```

```
      solution[0] = my_rank;
      place_queen(k, 1, my_rank,solution );

      MPI_Send(&Solutions, 1, MPI_INT, MASTER, 0, MPI_COMM_WORLD);
   }
   else {

      //master calculates a reduction
      for (source = 1; source < num_nodes; source++) {
         int response;
         MPI_Recv(&response, 1, MPI_INT, MPI_ANY_SOURCE, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("Process %d computed %d solutions\n", source,
             response);
         Solutions += response;
         printf("--------------------------------\n");

      }
   printf("Total Number of Solutions %d\n", Solutions);
   }

   MPI_Finalize();
   return 0;

}



void place_queen(int k, int col, int process, int * solution)
{
   //  srand(time(NULL));

   if (k == col)
   {

      Solutions++;

   }
   else{
```

```
    solution[0] = process;
    for(int row = 1; row <= k; row++)
    {
      // check if queen can be placed safely
      if(check_rows(row, col,solution) == 0 &&
          check_diagonals(row, col, solution) == 0)
      {
        //place queen on this row
        solution[col] = row;

        place_queen (k,col+1, process,solution);

      }//end inner if

    }//end loop
  }
}//end method

//will take a row and column and see if a queen is in danger
int check_rows(int row, int col, int * solution){
  //queens cant exist on the same row
  for(int i = 0; i < col; i ++){
    if(solution[i] == row ){
      return -1;
    }
  }
  return 0;
}


int check_diagonals(int row, int col, int * solution){
  //values we will be examining
  int danger1;
  int danger2;


  for(int i = 0; i < col; i ++){
    danger1 = solution[i]+i; // diagonals from left side
    danger2 = i - solution[i]; //diagonals from rights side
```

```
      if(row + col == danger1){ //left hand diagonals
         return -1;
      }
      if( col - row == danger2){ //right hand diagonals
         return -1;
      }
      }
   return 0;

}

//fill array with zeros
void initialize_array(int k, int * array){
   for (int i = 0; i < k; i++){
      array[i] = 0;
   }
}

//print array
void printArray(int k,int process, int * array){
   printf("PROCESS %d [", process);
   for (int i = 0; i < k; i++){
      printf("%d ", array[i]);
   }
   printf("]\n");

}
```