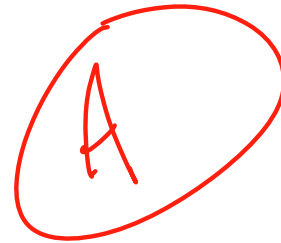


MPI-based Solutions to the k-Queens Problem
Programming Assignment #4
CS677 High Performance Computing
December 2, 2020
Andrew Weston



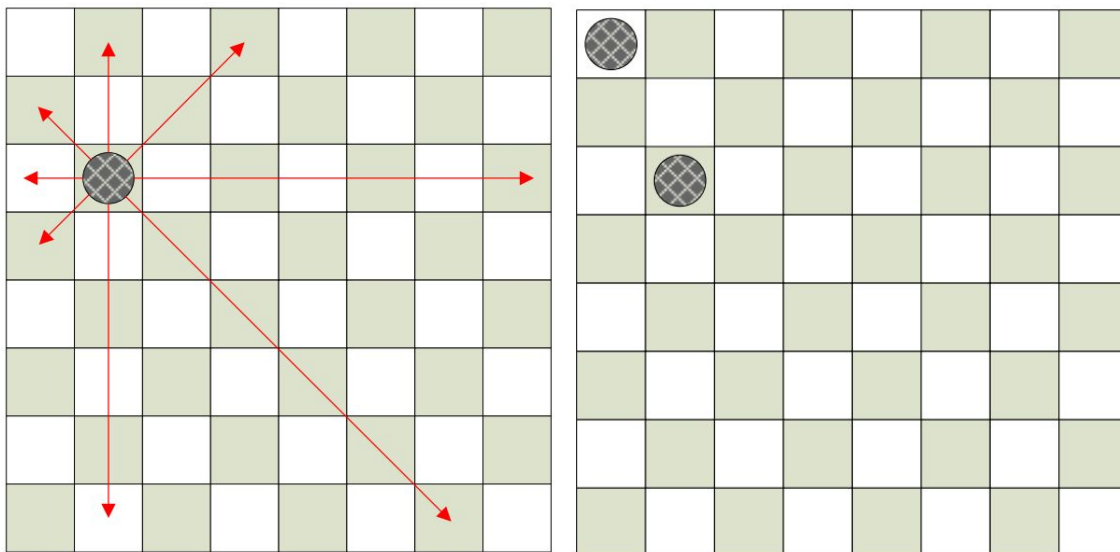
Abstract: The k -Queens problem is a search for valid arrangements of k queens on a $k \times k$ chessboard such that no two queens can attack each other. We present three parallel implementations based on an algorithm due to Wirth. We find that a parallel implementation using the Message-Passing Interface (MPI) can achieve a scalable Speedup of approximately 13x. We also demonstrate that an alternative implementation with limited scalability can achieve a 23x Speedup.

1. Introduction

1.1 k -Queens Problem

In the k -Queens problem, the goal is to place k queens on a $k \times k$ chessboard such that no two queens can attack each other. We remind the reader that queens can attack along their row, their column, and along each diagonal (see Figure 1 below left). Thus, a valid solution has no two queens in the same row, the same column, or on the same diagonal (see Figure 2 below right).

Figures 1 and 2



1.2 k -Queens Solutions

Solving this problem is essentially a search for valid solutions. One way to structure the approach and reduce the search space is to work across the board from the leftmost column using recursion where appropriate. Consider the traditional 8×8 chessboard and the classic 8-Queens problem in Figure 2 above. The first move is to place a queen in the first row of the

first column. Next, place the second queen anywhere in the second column such that it is “safe”. Note that it cannot be placed in row 1 or in row 2 (it is on the diagonal), but it can be safely placed in row 3. We continue until all 8 queens have been placed. If at any time a queen cannot be placed in a particular column, then we backtrack to the previous column and try placing its queen in a different row (i.e. backtrack up the search tree).

Queen placement can be simply and efficiently represented by indicating each queen’s row. For example, the sequence (1, 5, 8, 6, 3, 7, 2, 4) represents a valid solution, in which the queen in column 1 is located in row 1, the queen in column 2 is located in row 5, etc.

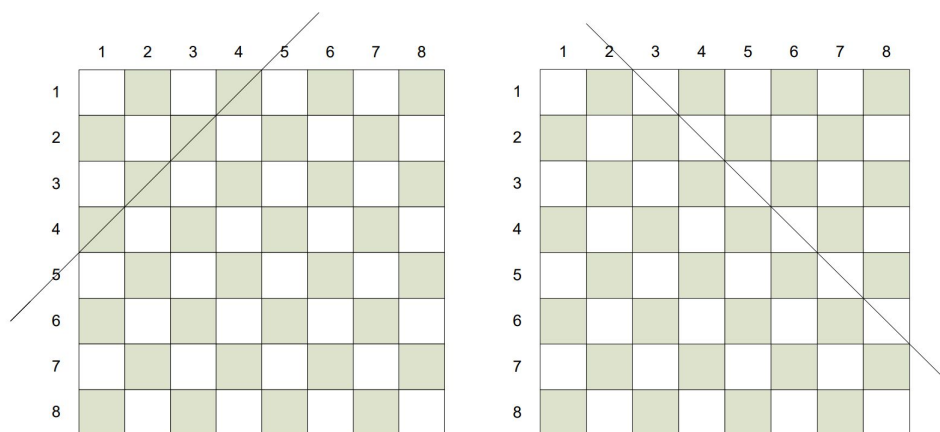
1.3 *k*-Queens Parallelization

This is a typical search problem. The straightforward approach involves generating potential solutions, evaluating each of them, and finally reporting valid results. The “tree” organization of the search space allows for a more directed search than brute-force computation. Using the message-passing paradigm implemented by the Message-Passing Interface(MPI) framework we can partition the workload, execute in parallel, and communicate the results.

1.4 Wirth’s Algorithm

Many of the solutions to the original 8-Queens problem use backtracking based on an algorithm due to Wirth. Clearly, two queens cannot be on the same row, and that is easy enough to check by maintaining a list of “placed” queen positions. Now, consider the diagonal as seen in Figure 3 (below left). Notice that each of the “threatened” squares can be identified as $rowNum + colNum = 5$. The Right-Hand Side (i.e. the sum) will differ, but the equality relation holds for all 15 possible diagonals of the same orientation. Therefore, similar to the row check, the legality of placing a queen in one of these diagonal positions can be checked by referring to the list of already placed queens. Similarly, consider the diagonal in Figure 4 (below to the right). Each of its “threatened” squares can be identified as $rowNum - colNum = -2$. Again, this relation holds for all diagonals of the same orientation and can be easily checked.

Figures 3 and 4



The pseudo code snippet in Figure 5 provides the structure of Wirth's algorithm. Iteration is used to examine all possible locations to place a queen. The conditional checks if a specific position is valid, and recursion is used to implement backtracking.

Figure 5

```
void place_queen (int col)
{
    if (all queens are placed) {
        Solutions++
    }
    else {
        // try all rows
        for (row=1; row <= k; row++) {
            // check if queen can be placed safely
            if (no queens on this row or on these diagonals)
                place queen on this row
            // try rest of problem
            place_queen (col+1);
            // returned from recursion
            // do not place queen on this row (backtrack)
        }
    }
}
```

2. Solutions using Wirth's Algorithm

First, we explain our sequential implementation of Wirth's algorithm. Next, we introduce an alternative sequential implementation as a foundation for one of our parallel approaches. Then, we consider our three parallel implementations. Thereafter, we discuss our solution validation strategy. Finally, we briefly discuss alternative implementations.

2.1 Sequential Solution

All of our implementations utilize the “iterative-recursive hybrid” structure introduced in Figure 5. We explain it in detail here and simply explain how it was applied to the other solutions in Sections 2.2-2.5. Our sequential implementation is a direct translation of the pseudo-code in Figure 5. Appendix A Line 131 shows how the seq_place_queen function takes a vector reference representing the list of queens' row assignments, the column to attempt to place a queen in, and a reference to the counter for valid solutions. The function handles two cases: the base case and the iterative-recursive case. In the base case, if the function is invoked with a column argument greater than k then we have placed k -Queens and found a valid solution. In the iterative-recursive case, we iteratively attempt to place the queen in each row of the specified column before recursing to the next column.

We rely on a helper function “attempt_placement” to check whether a placement is valid, update the queen row list, and implement indirect recursion. The attempt_placement helper function takes all of the parameters from seq_place queen: queen row list, column, and number of solutions. Additionally, it accepts a parameter *function* to be invoked as an indirect, recursive call if the prospective queen placement is valid.

We determine whether a prospective placement is valid as explained in Section 1.3. If a placement is valid, we update the queen row list (representing the placement of a queen) and use the function argument to recurse to the next column. The chief advantage of utilizing this attempt_placement helper function is that it simplifies three of our solutions allowing us to focus our analysis on contrasting their parallel structures.

2.2 “Prune” Sequential Solution

Next, we consider an extension to the baseline sequential solution in Section 2.1. Once we place a queen, we can immediately deduce the “threatened” positions represented by the red arrows in Figure 1. If we record the board positions that the placed queen threatens, then we have effectively pruned branches from the tree of possible board placements. Our implementation uses bit manipulation of a vector of integers to represent the board state. Each integer represents a row and each bit in the integer represents a square. Positions with a one are threatened. For example, Figure 6 (below left) shows the placement of a queen in column-one row-one for an 8x8 board (only the 8 most significant bits are actually utilized in the board representation). Figure 7 (below right) shows the same board after pruning the positions threatened by placing a queen in column-one row-five. Complete details of the pruned board state for the 8 x 8 solution sequence (1,5,8,6,3,7,2,4) are available in Appendix B.

Figures 6 and 7

```
1 11111111111111111111111111111111
2 11000000000000000000000000000000
3 10100000000000000000000000000000
4 10010000000000000000000000000000
5 10001000000000000000000000000000
6 10000100000000000000000000000000
7 10000010000000000000000000000000
8 10000001000000000000000000000000
9 10000000100000000000000000000000
10 10000000010000000000000000000000
11 10000000001000000000000000000000
12 10000000000100000000000000000000
13 10000000000010000000000000000000
14 10000000000001000000000000000000
15 10000000000000100000000000000000
16 10000000000000010000000000000000
17 10000000000000001000000000000000
18 10000000000000000100000000000000
19 10000000000000000010000000000000
20 10000000000000000001000000000000
21 10000000000000000000100000000000
22 10000000000000000000010000000000
23 10000000000000000000001000000000
24 10000000000000000000000100000000
25 10000000000000000000000010000000
26 10000000000000000000000001000000
27 10000000000000000000000000100000
28 10000000000000000000000000010000
29 10000000000000000000000000001000
30 10000000000000000000000000000100
31 10000000000000000000000000000010
32 10000000000000000000000000000001
34 11111111111111111111111111111111
35 11001000000000000000000000000000
36 11110000000000000000000000000000
37 11110000000000000000000000000000
38 11111111111111111111111111111111
39 11100100000000000000000000000000
40 11010010000000000000000000000000
41 11001001000000000000000000000000
42 11000100100000000000000000000000
43 11000010010000000000000000000000
44 11000001001000000000000000000000
45 11000000100100000000000000000000
46 11000000010010000000000000000000
47 11000000001001000000000000000000
48 11000000000100100000000000000000
49 11000000000010010000000000000000
50 11000000000001001000000000000000
51 11000000000000100100000000000000
52 11000000000000010010000000000000
53 11000000000000001001000000000000
54 11000000000000000100100000000000
55 11000000000000000010010000000000
56 11000000000000000001001000000000
57 11000000000000000000100100000000
58 11000000000000000000010010000000
59 11000000000000000000001001000000
60 11000000000000000000000100100000
61 11000000000000000000000010010000
62 11000000000000000000000001001000
63 11000000000000000000000000100100
64 11000000000000000000000000010010
65 11000000000000000000000000001001
```



The `seq_prune_queen` implementation from Appendix A Line 157 is substantially similar to the sequential solution in Section 2.1 except for two key differences. First, it expands on the parameter list to include a pointer to a vector of integers representing the board state. Second, in the recursive case, rather than using the `attempt_placement` helper function this solution simply checks the bit representing the board position to determine whether the position is threatened or not. If the position is not threatened, we copy the board state, place the queen (by pruning the positions it threatens), and recurse.

The advantage of this approach is that we avoid recomputation of “pruned” board positions using efficient bitwise operations. The disadvantage of this approach is that it requires $O(k^2)$ integers of memory to concurrently maintain the board state for all k columns to reach the base case.

Notably we would normally include only the best-known sequential algorithm, but we choose to explain both the straightforward sequential approach and this sequential prune approach to support our explanation in Section 2.5 and our analysis in Section 3. By including both implementations, we can separate and analyze the speedup due to using the faster pruning algorithm from the speedup due to parallelism.

2.3 k -Partitions Parallel Solution

One straightforward parallel approach is “parallelizing” the sequential solution in Section 2.1. Using k MPI processes, we divide the search space so that each process i begins by placing its first queen in row i and continues its search in the next column. In this way, the processes work in parallel to evaluate different branches of the tree. Once all of the processes are complete, a simple `MPI_Reduce()` command can aggregate local solutions into a cluster-wide, global, solutions count. The advantage of this approach is that it parallelizes the workload and communicates results *simply*. Its primary disadvantage is that it can only utilize k degrees of parallelism.

2.4 k^2 -Partitions Parallel Solution

How can we maintain the simple partitioning and reduction of the k -Partitions solution, but support a higher degree of parallelism?

In the k -Partitions solution, we effectively divided the k -ary search tree so that each process took one of the k subtrees. Assuming k is large enough, we can extend this idea to arbitrary depth in the search tree. For our purposes, it is sufficient to note that the first level of the k -ary search tree has k subtrees and the second level has k^2 subtrees. Assigning processes' search space by second level subtrees would allow us to utilize k^2 degrees of parallelism.

In our implementation, we leave the base case unchanged and modify the iterative-recursive case to distinguish between whether or not we are evaluating the first column. If we are evaluating the first column, then each of the k^2 processes independently generates all of the potential queen placements. Thereafter, the processes divide the workload of searching the second level subtrees based on their respective ranks.

We briefly note that our k^2 solution has been validated for $k > 1$. In the case of $k = 1$, our solution is invalid because of how each process redundantly computes the first column before the global reduction. Handling this case without affecting the computation when $k > 1$ was nontrivial.


The advantage of this approach is that it supports a high degree of parallelism. As k increases the number of processes we can utilize grows quadratically. The disadvantage of this implementation is the required overhead to redundantly compute the first column in every process and to reduce the solutions generated in all k^2 processes.

2.5 “Prune” Parallel Solution

We can combine the pruning approach in Section 2.2 with the k^2 partitioning in Section 2.4 into a hybrid approach: k^2 processes each prune their respective subtrees.

The advantages of this approach are that it supports a high degree of parallelism and efficient bitwise pruning. Like the sequential prune solution, the disadvantage of this approach is that it requires $O(k^2)$ integers of memory to maintain the board state for all k columns concurrently to reach the base case.


2.6 Solution Validation

First, we developed the straightforward sequential solution explained in Section 2.1. We manually confirmed that the six smallest boards, $1 \leq k \leq 6$, produced valid arrangements. Then (once the number of solutions became cumbersome) we ran our sequential program for $1 \leq k \leq 16$. We validated that our sequential implementation computed the proper number of solutions and manually verified a sample of the results[1]. Complete details of the 92 solutions for the traditional 8 x 8 board are available in Appendix C. Similarly, complete details for the total number of solutions for different values of k are available in Appendix D. 

We validated our parallel solutions using the six smallest boards, $1 \leq k \leq 6$ as with our sequential validation. However, thereafter when the number of solutions became cumbersome we simply validated the total number of solutions. We would expect the order of solutions to be nondeterministic so further validation was impractical. A sample BASH script for validating our parallel programs' total number of solutions is included in Appendix E.

2.7 Alternative Solutions

One logical extension to the k^2 -Partitions Parallel Solution in Section 2.4 is to dynamically tailor the algorithm to redundantly compute the first “depth” columns where $depth = \log_k(\text{processes})$. However, for large values of k , the maximum number of available processes the cluster could support would probably be the limiting factor. Thus, the performance impact of allowing k^2 versus k^{depth} processes would probably not be significant.

More importantly, a natural extension to the “Prune” Parallel Solution in Section 2.5 is to dynamically limit the depth of the recursion that a single process will perform to that of its available memory and then make a recursive call to another process via message-passing “passing the baton”. With this approach, a process' depth of recursive calls is limited by how many board copies it can instantiate. When one process is nearing its memory limit, the process could instead send a message with the recursive call to another process. In this way, the limit on the depth of recursion would be the sum of the memory spaces of the whole virtual cluster. 

3. Experimental Results

We determined experimental results on a Grand Valley State University Architecture Laboratory virtual cluster. The system included 10 machines - each with an AMD Ryzen 7 2700x 8-core CPU (16 logical cores) with 16 GB RAM. Additionally, every machine ran Ubuntu 20.04. Our trial BASH script and full details of every trial's performance are available in Appendix F and G respectively.

Additionally, we note that Speedup is defined by the equation:

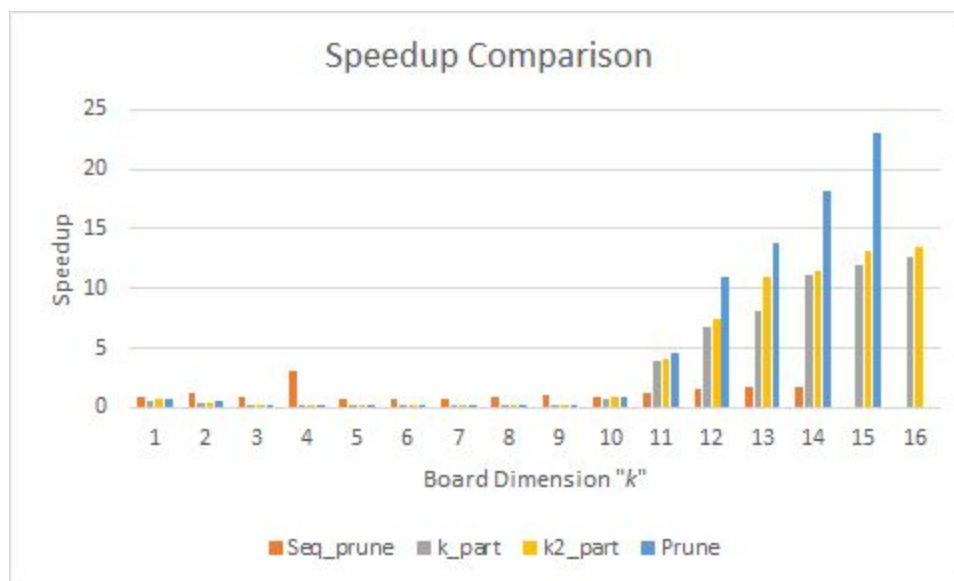
$$Speedup = Time_{Sequential} / Time_{Parallel}$$

We ran two iterations of each algorithm for the range of k -values: $1 \leq k \leq 16$. In the first iteration, we constrained our parallel solutions to use no more than 80 MPI processes. In the second iteration, we allowed our parallel solutions to use 160 MPI processes (the maximum supported by the virtual cluster).

All of our Speedup calculations utilize the sequential algorithm explained in Section 2.1 rather than the alternative pruning algorithm in Section 2.2 which suffered from scalability limitations.

Figure 8 and 9 below show how the parallel prune solution using 80 MPI processes achieved the highest overall Speedup of 23x for $k = 15$. In general, the parallel prune solution consistently achieved the best Speedup once the algorithms' performance began to materially diverge when k was greater than 10. However a key limitation of this algorithm is that it could not compute k greater than 15 because of the space required to support the board state pruning.

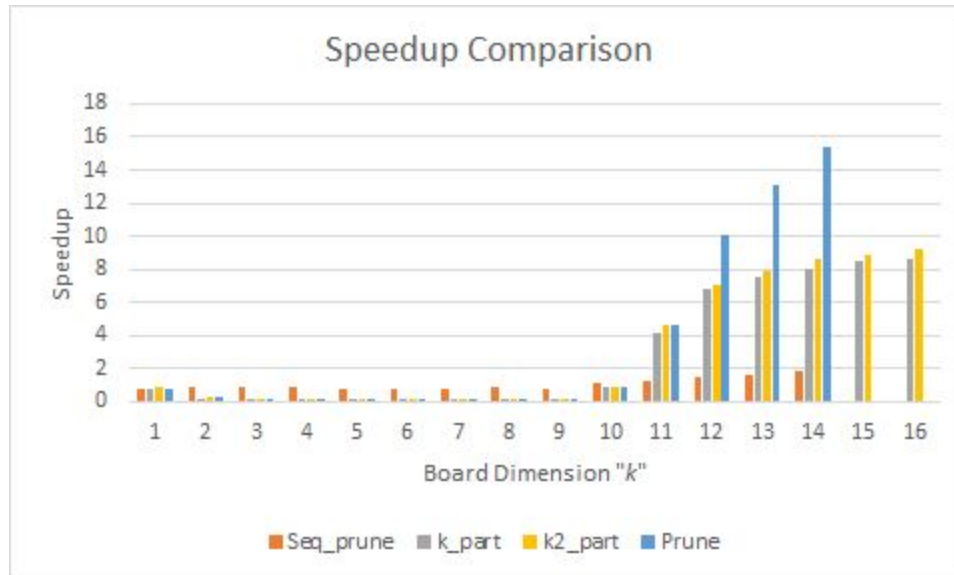
Figure 8



In contrast to Figure 8, Figure 9 shows how using the maximum number of MPI processes, 160, actually decreased Speedup. This is probably due to contention between processes to

access a physical core.

Figure 9

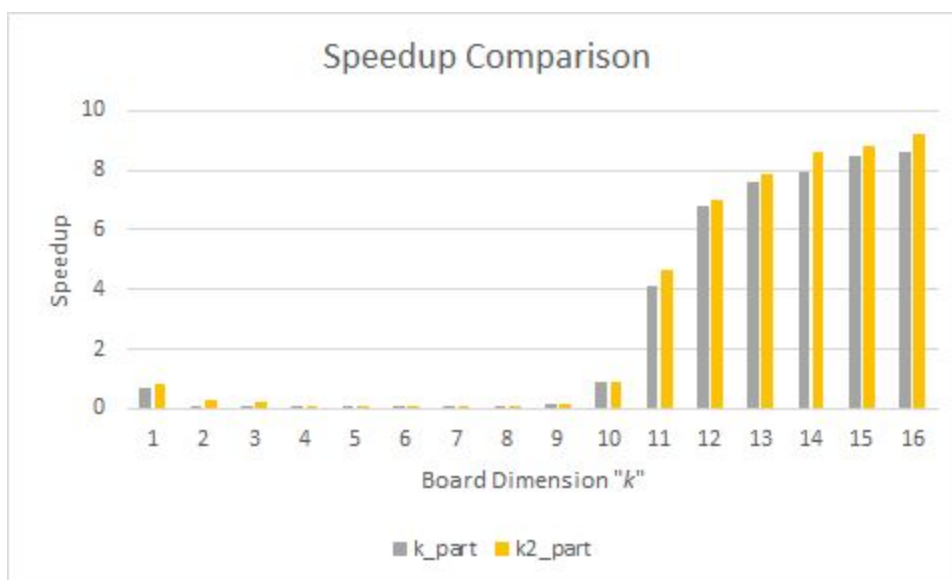


Additionally, both the k and k^2 partitioning algorithms achieved a Speedup of approximately 12x for relatively large values of k in a reasonably scalable manner. Narrowing our focus to after the runtimes began to diverge at $k = 10$, Figure 10 emphasizes the relative differences in Speedup between the two algorithms. Since this trial was limited to a total of 80 MPI processes, the k^2 solution does not benefit from any additional parallelism after $k = 13$ where we saw the largest relative difference in Speedup. Thereafter, both solutions had similar Speedup. This trend was consistent across all values of k during the second iteration using 160 MPI processes (see Figure 11).

Figure 10



Figure 11



The commiserate performance data shows a consistent stratification of solutions as k varies with both 80 and 160 MPI processes (see Figures 12 and 13 respectively). Note: the k -partitions performance is presented, but is largely obscured by the similar k^2 -partitions data.

Figure 12

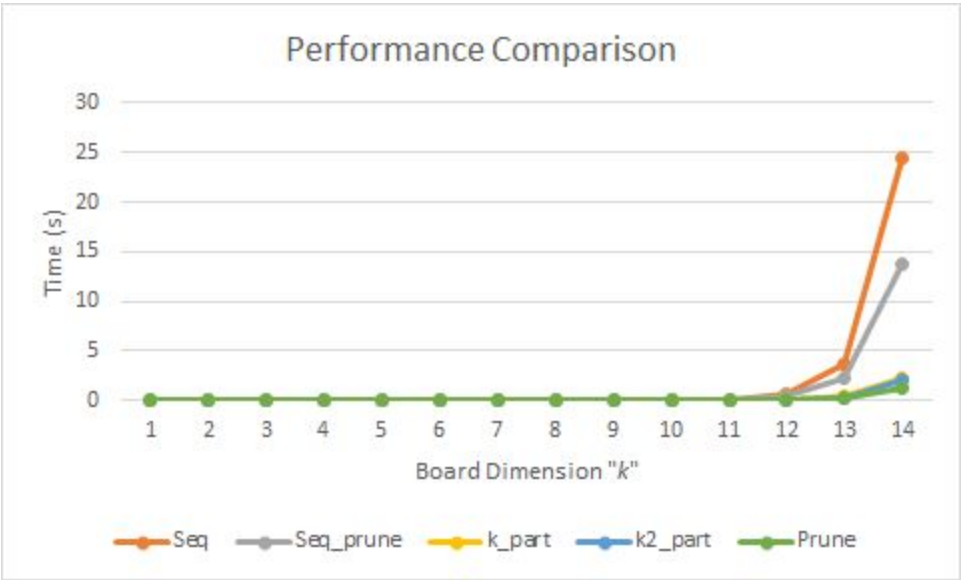
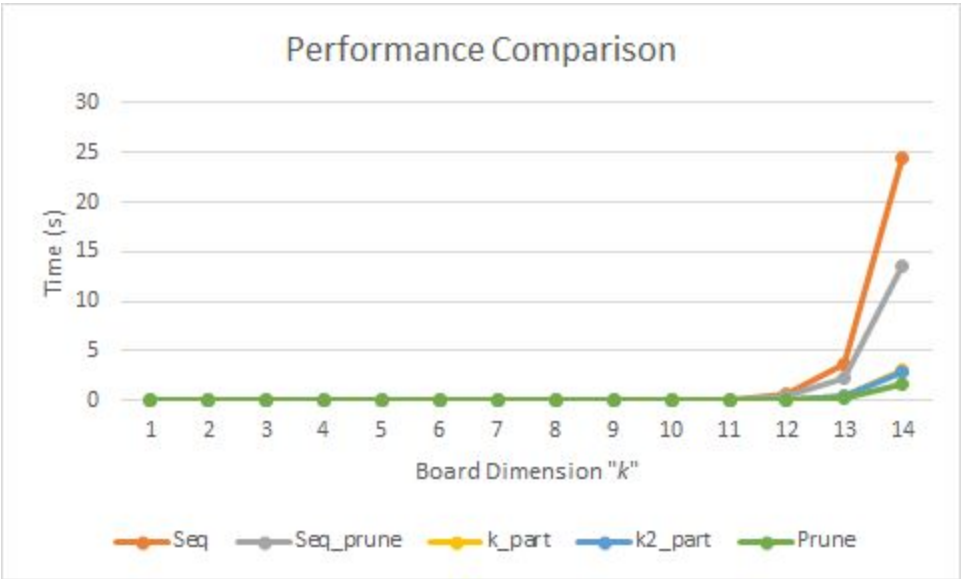


Figure 13



4. Problems and Solutions

4.1 MPI Communication - This was my first project using MPI beyond simple exercises to understand the framework's essential constructs. My intuition was that limiting communication to a simple pattern would make synchronization (and therefore debugging) easier. I think this was a sensible approach but I regret not exercising other MPI constructs in different parallel approaches.

Specifically, I regret not implementing a parallel pruning implementation that would pass a message representing a recursive call to another process. I think it would have been feasible and faster than anything else I implemented, but I did not realize it until shortly before the project was due. I was limited by thinking of an MPI program as effectively a multi-threaded program, but one advantage of programming a cluster is the flexibility to access more aggregate memory space to support deeper recursion. It should have dawned on me that when one process is running out of memory due to recursion it should send a message "asking for help".

4.2 C++ Memory Management

I spent a significant amount of time developing the bit vector "pruning" approach because I thought it would be faster than the straightforward sequential approach by reducing redundant checks for whether a position was threatened. I had a time-consuming bug where my program would work while $k \leq 6$ but would crash for larger boards. This suggested that I had a memory issue but the MPI error message was unclear. I believe the problem was that I was allocating the board copies on the stack and the program would crash once a page was full or possibly when the stack's partition was exceeded. Ultimately, allocating the vectors on the heap and passing around their pointers worked well.

References

[1] A000170. (n.d.). Retrieved December 02, 2020, from <https://oeis.org/A000170>

(+) VALIDATED RESULTS
EXCELLENT SPEEDUPS
CREATIVE ALTERNATIVE
IMPLEMENTATIONS
PROFESSIONALLY - WRITTEN
REPORT

EXCELLENT IO3

```

1 #include <iostream>
2 #include <vector>
3 #include <mpi.h>
4 #include <bitset>
5
6 using namespace std;
7
8 /* Implementation of Wirth's Algorithm */
9 void seq_place_queen(vector<int> &queen_row_list, int col, int &solutions);
10
11 /* Implementation of Wirth's Algorithm with bitwise branch pruning */
12 void seq_prune_queen(vector<int> &queen_row_list, vector<int> *board, int col,
    int &solutions);
13
14 /* Parallel Implementation of Wirth's Algorithm */
15 void k_partitions(vector<int> &queen_row_list, int col, int &solutions);
16
17 /* Parallel Implementation of Wirth's Algorithm */
18 void k2_partitions(vector<int> &queen_row_list, int col, int &solutions);
19
20 /* Parallel Implementation of Wirth's Algorithm with bitwise branch pruning */
21 void prune_parallel_place_queen(vector<int> &queen_row_list, vector<int> *board
    , int col, int &solutions);
22
23 /* Helper function to check whether a queen can be placed */
24 void attempt_placement(vector<int> &queen_row_list, int col, int &solutions,
    int row,
25
    void fun(vector<int> &queen_row_list, int col, int &
    solutions));
26
27 /* Helper function to print the list of placed queens */
28 void print_queen_row_list(const vector<int> &queen_row_list);
29
30 /* Helper function prune "threatened" board positions */
31 void prune_board(vector<int> *board, int k, int row, int col);
32
33 /* Helper function to print the bit vector board state */
34 void print_board(vector<int> *board, int k);
35
36 /* Helper function to get the state of the bit at index i */
37 bool get_bit(int n, int i);
38
39 /* Helper function to set the state of the bit at index i */
40 int set_bit(int n, int i);
41
42 /* The size of one dimension of the chess board */
43 int K = 0;
44
45 /* The number of cooperating MPI processes */
46 int size = 0;
47
48 /* The rank (ID) of a particular MPI process */
49 int ps_rank = 0;
50
51 /* The rank of a manager process */
52 const int MANAGER = 0;
53
54 /* The first column */
55 const int FIRST_COL = 1;
56
57 /* The MPI_INT elements to reduce */
58 const int SOLUTIONS_VAR_SIZE = 1;
59

```



```

60 /* The number of bits in a four byte integer */
61 const int BITS_PER_INT = 32;
62
63
64 /******
65 * This function serves as the program "driver" for an implementation
66 * of several solutions to the k-Queens problem using MPI. It takes
67 * command line arguments for the desired algorithm and the problem
68 * size (k), uses an iterative-recursive algorithm to solve the problem
69 * and optionally prints results.
70 *
71 * @param argc is the number of command-line arguments.
72 * @param v is the array of strings representing the command-line args.
73 *****
74 int main(int argc, char *argv[]) {
75     if (argc < 3) {
76         cerr << "Invalid Use\n Syntax is: algorithm k_queens" << endl;
77         cerr << "Algorithms are seq, k_partitions"
78             << endl;
79     }
80     MPI_Init(&argc, &argv);
81     double start = MPI_Wtime();
82     string algorithm(argv[1]);
83     K = stoi(argv[2]);
84     vector<int> queen_row_list;
85     int ps_solutions = 0;
86     int total_solutions = 0;
87
88     //seq programs don't require a reduce bc they only use one ps
89     if (algorithm == "seq") {
90         seq_place_queen(queen_row_list, FIRST_COL, ps_solutions);
91         total_solutions = ps_solutions;
92     } else if (algorithm == "seq_prune") {
93         auto *board = new vector<int>(K);
94         seq_prune_queen(queen_row_list, board, FIRST_COL,
95             ps_solutions);
96         total_solutions = ps_solutions;
97     } else if (algorithm == "k_partitions") {
98         k_partitions(queen_row_list, FIRST_COL, ps_solutions);
99         MPI_Reduce(&ps_solutions, &total_solutions, SOLUTIONS_VAR_SIZE,
100             MPI_INT, MPI_SUM, MANAGER, MPI_COMM_WORLD);
101     } else if (algorithm == "k2_partitions") {
102         k2_partitions(queen_row_list, FIRST_COL, ps_solutions);
103         MPI_Reduce(&ps_solutions, &total_solutions, SOLUTIONS_VAR_SIZE,
104             MPI_INT, MPI_SUM, MANAGER, MPI_COMM_WORLD);
105     } else if (algorithm == "prune") {
106         auto *board = new vector<int>(K);
107         prune_parallel_place_queen(queen_row_list, board, FIRST_COL,
108             ps_solutions);
109         MPI_Reduce(&ps_solutions, &total_solutions, SOLUTIONS_VAR_SIZE,
110             MPI_INT, MPI_SUM, MANAGER, MPI_COMM_WORLD);
111     }
112     double end = MPI_Wtime();
113     if (ps_rank == MANAGER) {
114         cout << "Total solutions: " << total_solutions << endl;
115         printf("%f\n", end - start);
116     }
117     MPI_Finalize();
118     return 0;
119 }
120
121
122 /******

```

```

123 * This helper function serves as the program implementation
124 * of an iterative-recursive solution to the k-Queens problem using
125 * a single MPI process "sequential" algorithm.
126 *
127 * @param queen_row_list is the list of placed queens.
128 * @param col is the column to attempt to place a queen in.
129 * @param solutions is a reference to the counter for valid solutions.
130 *****/
131 void seq_place_queen(vector<int> &queen_row_list, int col,
132                    int &solutions) {
133     if (col > K) {
134         //print_queen_row_list(queen_row_list);
135         solutions++;
136     } else {
137         for (int row = 1; row <= K; row++) {
138             attempt_placement(queen_row_list,
139                             col, solutions, row,
140                             seq_place_queen);
141         }
142     }
143 }
144
145
146 /*****
147 * This helper function serves as the program implementation
148 * of an iterative-recursive solution to the k-Queens problem using
149 * a single MPI process "sequential" algorithm that incorporates
150 * pruning of a bit vector.
151 *
152 * @param queen_row_list is the list of placed queens.
153 * @param board is the board state of threatened positions.
154 * @param col is the column to attempt to place a queen in.
155 * @param solutions is a reference to the counter for valid solutions.
156 *****/
157 void seq_prune_queen(vector<int> &queen_row_list, vector<int> *board,
158                    int col, int &solutions) {
159     if (col > K) {
160         //print_queen_row_list(queen_row_list);
161         solutions++;
162     } else {
163         for (int row = 1; row <= K; row++) {
164             if (!get_bit((*board)[row - 1], BITS_PER_INT - col)) {
165                 auto *board_cp = new vector<int>(*board);
166                 prune_board(board_cp, K, row, col);
167                 prune_parallel_place_queen(queen_row_list, board_cp,
168                                         col + 1, solutions);
169             }
170         }
171     }
172 }
173
174
175 /*****
176 * This helper function serves as the program implementation
177 * of an iterative-recursive solution to the k-Queens problem using
178 * parallel MPI processes.
179 *
180 * @param queen_row_list is the list of placed queens.
181 * @param col is the column to attempt to place a queen in.
182 * @param solutions is a reference to the counter for valid solutions.
183 *****/
184 void k_partitions(vector<int> &queen_row_list, int col,
185                 int &solutions) {

```

```

186     MPI_Comm_size(MPI_COMM_WORLD, &size);
187     MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
188     if (col > K) {
189         //print_queen_row_list(queen_row_list);
190         solutions++;
191     } else {
192         //assign level 0 k-ary tree search by number of processes
193         if (col == 1) {
194             for (int row = 1 + ps_rank; row <= K; row += size) {
195                 attempt_placement(queen_row_list, col, solutions, row,
196                                 k_partitions);
197             }
198         } else {
199             for (int row = 1; row <= K; row++) {
200                 attempt_placement(queen_row_list, col, solutions, row,
201                                 k_partitions);
202             }
203         }
204     }
205 }
206
207
208 /*****
209  * This helper function serves as the program implementation
210  * of an iterative-recursive solution to the k-Queens problem using
211  * parallel MPI processes.
212  *
213  * @param queen_row_list is the list of placed queens.
214  * @param col is the column to attempt to place a queen in.
215  * @param solutions is a reference to the counter for valid solutions.
216  *****/
217 void k2_partitions(vector<int> &queen_row_list, int col,
218                  int &solutions) {
219     MPI_Comm_size(MPI_COMM_WORLD, &size);
220     MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
221     if (col > K) {
222         //print_queen_row_list(queen_row_list);
223         solutions++;
224     } else {
225         const int K_SQ = K * K;
226         if (col == 1) {
227             if (ps_rank < K_SQ) {
228                 for (int pseudo_row = 1; pseudo_row <= K;
229                     pseudo_row++) {
230                     queen_row_list.push_back(pseudo_row);
231                     for (int row = 1 + ps_rank; row <= K;
232                         row += K_SQ) {
233                         attempt_placement(queen_row_list, col + 1,
234                                         solutions, row,
235                                         k2_partitions);
236                     }
237                     queen_row_list.pop_back();
238                 }
239             }
240         } else {
241             for (int row = 1; row <= K; row++) {
242                 attempt_placement(queen_row_list, col, solutions,
243                                 row, k2_partitions);
244             }
245         }
246     }
247 }
248

```

```

249
250 /*****
251  * This helper function serves as the program implementation
252  * of an iterative-recursive solution to the k-Queens problem using
253  * a parallel MPI processes that incorporate pruning of a bit vector.
254  *
255  * @param queen_row_list is the list of placed queens.
256  * @param board is the board state of threatened positions.
257  * @param col is the column to attempt to place a queen in.
258  * @param solutions is a reference to the counter for valid solutions.
259  *****/
260 void prune_parallel_place_queen(vector<int> &queen_row_list,
261                                vector<int> *board, int col,
262                                int &solutions) {
263     MPI_Comm_size(MPI_COMM_WORLD, &size);
264     MPI_Comm_rank(MPI_COMM_WORLD, &ps_rank);
265     if (col > K) {
266         //print_queen_row_list(queen_row_list);
267         solutions++;
268     } else {
269         const int K_SQ = K * K;
270         if (col == FIRST_COL) {
271             if (ps_rank < K_SQ) {
272                 for (int pseudo_row = 1; pseudo_row <= K;
273                     pseudo_row++) {
274                     queen_row_list.push_back(pseudo_row);
275                     auto *pseudo_board = new vector<int>(*board);
276                     prune_board(pseudo_board, K, pseudo_row, col);
277                     for (int row = 1 + ps_rank; row <= K;
278                         row += K_SQ) {
279                         // -1s to convert to equivalent index
280                         if (!get_bit((*pseudo_board)[row - 1],
281                                     BITS_PER_INT - col - 1)) {
282                             auto *board_cp =
283                                 new vector<int>(*pseudo_board);
284                             queen_row_list.push_back(row);
285                             prune_board(board_cp, K, row, col + 1);
286                             // +2 = +1 to advance to next column and
287                             // +1 for the first col
288                             // (first col computed with outer loop)
289                             prune_parallel_place_queen(queen_row_list,
290                                                         board_cp,
291                                                         col + 1 + 1,
292                                                         solutions);
293                             queen_row_list.pop_back();
294                         }
295                     }
296                     queen_row_list.pop_back();
297                 }
298             }
299         } else {
300             for (int row = 1; row <= K; row++) {
301                 if (!get_bit((*board)[row - 1], BITS_PER_INT - col)) {
302                     auto *board_cp = new vector<int>(*board);
303                     prune_board(board_cp, K, row, col);
304                     prune_parallel_place_queen(queen_row_list,
305                                                 board_cp, col + 1,
306                                                 solutions);
307                 }
308             }
309         }
310     }
311 }

```

```

312
313
314 /*****
315  * This helper function prints the queen row list representing the
316  * current placement of queens.
317  *
318  * @param queen_row_list is the list of placed queens.
319  *****/
320 void print_queen_row_list(const vector<int> &queen_row_list) {
321     for (auto el: queen_row_list)
322         cout << el << ' ';
323     cout << endl;
324 }
325
326
327 /*****
328  * This helper function attempts to place a queen in the specified
329  * position and recurse to search for valid queen placements in the
330  * the succeeding columns.
331  *
332  * @param queen_row_list is the list of placed queens.
333  * @param col is the column to attempt to place a queen in.
334  * @param solutions is a reference to the counter for valid solutions.
335  * @param row is the row to attempt to place a queen in.
336  * @param fun is the function to recursively call.
337  *****/
338 void attempt_placement(vector<int> &queen_row_list, int col,
339                       int &solutions, int row,
340                       void fun(vector<int> &queen_row_list, int col,
341                               int &solutions)) {
342     bool canPlace = true;
343     //check the entire list to see if this row is safe
344     for (long unsigned int c = 1;
345          c <= queen_row_list.size() && canPlace; c++) {
346         if (queen_row_list[c - 1] == row)
347             canPlace = false;
348         // row + col
349         if (queen_row_list[c - 1] + c == row + col)
350             canPlace = false;
351         // row - col
352         if (queen_row_list[c - 1] - c == row - col)
353             canPlace = false;
354     }
355     if (canPlace) {
356         //place
357         queen_row_list.push_back(row);
358         //descend this subtree
359         fun(queen_row_list, col + 1, solutions);
360         //reset so iteration will advance to next subtree
361         queen_row_list.pop_back();
362     }
363 }
364
365
366 /*****
367  * This helper function "prunes" the parameter bit vector board state
368  * by setting "threatened" positions.
369  *
370  * @param board is the board state of threatened positions.
371  * @param row is the row to attempt to place a queen in.
372  * @param col is the column to attempt to place a queen in.
373  * @param k is a single dimension of the board.
374  *****/

```

```

375 void prune_board(vector<int> *board, int k, int row, int col) {
376     int row_index = row - 1;
377     int col_index = col - 1;
378
379     //prune row
380     (*board)[row - 1] = (~0);
381     //prune col
382     for (int m = 0; m < k; m++) {
383         (*board)[m] |= set_bit((*board)[m], BITS_PER_INT - col);
384     }
385     // k-1 because last column start is just row
386     int diag1_starting_row_index = max(0, (row_index - (k - 1 - col_index)));
387     int diag1_starting_col_index = min(row_index + col_index, k - 1);
388
389     //diags have increasing then decreasing size with max of k
390     int diag1_length = min((row + col - 1), k);
391     for (int l = 0; l < diag1_length && (diag1_starting_row_index + l) < k; l
392 ++) {
393         (*board)[diag1_starting_row_index + l] |=
394             set_bit((*board)[diag1_starting_row_index + l],
395                 BITS_PER_INT - diag1_starting_col_index - 1 + l);
396     }
397     int diag2_starting_row_index = row - min(row, col);
398     int diag2_starting_col_index = col - min(row, col);
399     int diag2_length = k - abs(row - col);
400     for (int l = 0; l < diag2_length; l++) {
401         (*board)[diag2_starting_row_index + l] |= set_bit((*board)[
402             diag2_starting_row_index + l],
403                 BITS_PER_INT -
404             diag2_starting_col_index - l - 1);
405     }
406 }
407
408 /*****
409  * This helper function prints the bit vector board state.
410  *
411  * @param board is the board state of threatened positions.
412  * @param k is a single dimension of the board.
413  *****/
414 void print_board(vector<int> *board, int k) {
415     for (int r:(*board)) {
416         bitset<32> row(r);
417         cout << row << endl;
418     }
419     cout << endl;
420 }
421
422 /*****
423  * This helper function sets the specified index bit of the specified
424  * number.
425  *
426  * @param n is the bit vector to be updated.
427  * @param i is the index of the bit position to be set.
428  *
429  * @return is the updated bit vector.
430  *****/
431 int set_bit(int n, int i) {
432     return (n | (1 << i));
433 }
434

```



```
435
436 /*****
437  * This helper function gets the specified index bit of the specified
438  * number.
439  *
440  * @param n is the bit vector to be retrieved from.
441  * @param i is the index of the bit position to be retrieved.
442  *
443  * @return is the bool representation of the specified bit.
444  *****/
445 bool get_bit(int n, int i) {
446     return ((n & (1 << i)) != 0);
447 }
448
```

Appendix B - *k*-Queens “Pruning” Solution Sample Board States

[illegible]

1100000000000000001001000000000000
1100000000000000000100100000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000
1100000000000000000010010000000000

1111111111111111111111111111111111
1110100010000000000000000000000000
1111000100000000000000000000000000
1111001000000000000000000000000000
1111111111111111111111111111111111
1110110000000000000000000000000000
1111001000000000000000000000000000
1111111111111111111111111111111111
1111010010000000000000000000000000
1110101001000000000000000000000000
1110010100100000000000000000000000
1110001010010000000000000000000000
1110000101001000000000000000000000
1110000010100100000000000000000000
1110000001010010000000000000000000
1110000000101001000000000000000000
1110000000010100100000000000000000
1110000000001010010000000000000000
1110000000000101001000000000000000
1110000000000010100100000000000000
1110000000000001010010000000000000
1110000000000000101001000000000000
1110000000000000010100100000000000
1110000000000000001010010000000000
1110000000000000000101001000000000
1110000000000000000010100100000000
1110000000000000000001010010000000
1110000000000000000000101001000000
1110000000000000000000010100100000
1110000000000000000000001010010000
1110000000000000000000000101001000
1110000000000000000000000010100100
1110000000000000000000000001010010
1110000000000000000000000000101001

1111111111111111111111111111111111
1111100110000000000000000000000000
1111001100000000000000000000000000
1111011000000000000000000000000000
1111111111111111111111111111111111
1111111111111111111111111111111111
1111101000000000000000000000000000
1111111111111111111111111111111111


```
111110000000000000000000010110101  
11111000000000000000000000001011010  
11111000000000000000000000000101101
```

```

111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111101000000000000000000000000000
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111111111111111111111111111111
111111111101001000000000000000000000
111111111110100100000000000000000000
111111111110100100000000000000000000
111111101111010010000000000000000000
111111101111010010000000000000000000
111111101011110100100000000000000000
111111100101111010010000000000000000
111111100010111101001000000000000000
111111100001011110100100000000000000

```


Appendix C - Solutions to the 8-Queens Problem

15863724
16837425
17468253
17582463
24683175
25713864
25741863
26174835
26831475
27368514
27581463
28613574
31758246
35281746
35286471
35714286
35841726
36258174
36271485
36275184
36418572
36428571
36814752
36815724
36824175
37285146
37286415
38471625
41582736
41586372
42586137
42736815
42736851
42751863
42857136
42861357
46152837
46827135
46831752
47185263
47382516
47526138
47531682
48136275
48157263
48531726
51468273
51842736
51863724
52468317
52473861
52617483
52814736

53168247
53172864
53847162
57138642
57142863
57248136
57263148
57263184
57413862
58413627
58417263
61528374
62713584
62714853
63175824
63184275
63185247
63571428
63581427
63724815
63728514
63741825
64158273
64285713
64713528
64718253
68241753
71386425
72418536
72631485
73168524
73825164
74258136
74286135
75316824
82417536
82531746
83162574
84136275

Appendix D - Number of Solutions to the k -Queens Problem (1 - 16)

1	Total solutions:	1
2	Total solutions:	0
3	Total solutions:	0
4	Total solutions:	2
5	Total solutions:	10
6	Total solutions:	4
7	Total solutions:	40
8	Total solutions:	92
9	Total solutions:	352
10	Total solutions:	724
11	Total solutions:	2680
12	Total solutions:	14200
13	Total solutions:	73712
14	Total solutions:	365596
15	Total solutions:	2279184
16	Total solutions:	14772512

Appendix E - Sample Validation BASH Script

```
1 #!/bin/bash
2 rm kq ./test_bench/solutions/seq_prune.txt
3 mpiCC main.cpp -o kq
4 if [ $? -eq 0 ]
5 then
6     clear;
7     for k in {1..15}
8     do
9         mpirun -np 1 --host arch06 ./kq seq_prune $k >> ./test_bench/solutions/${k}
10        seq_prune.txt
11        diff ./test_bench/solutions/seq.txt ./test_bench/solutions/${k}seq_prune.txt
12    done
13 fi
```


Appendix F - Sample Trials BASH Script

(160 process trials performed at off-peak time early in the morning)

```
1#!/bin/bash
2rm kq
3mpiCC main.cpp -o kq
4if [ $? -eq 0 ]
5then
6  for k in {1..16}
7  do
8    echo $k
9    k_sq=$((k*k))
10   mpirun -np 1 --host arch06 ./kq seq $k >> ./results/seq.txt
11   mpirun -np 1 --host arch06 ./kq seq_prune $k >> seq_prune.txt
12   mpirun -np $((($k_sq>160 ? 160 : $k_sq)) --hostfile my_hosts_16per ./kq k_partitions $k >> ./results/k.txt
13   mpirun -np $((($k_sq>160 ? 160 : $k_sq)) --hostfile my_hosts_16per ./kq k2_partitions $k >> ./results/k2.txt
14   mpirun -np $((($k_sq>160 ? 160 : $k_sq)) --hostfile my_hosts_16per ./kq prune $k >> ./results/prune.txt
15  done
16fi
```

Appendix G - Trial Results

(blank fields represent program failing to compute the solution due to memory limitations)

80 MPI Processes Trials					
k	Seq	Seq_prune	k_part	k2_part	Prune
1	0.000012	0.000014	0.000021	0.000015	0.000018
2	0.000016	0.000014	0.000041	0.00004	0.000027
3	0.000015	0.000018	0.020295	0.020311	0.020293
4	0.000045	0.000015	0.020307	0.020304	0.020268
5	0.000024	0.000031	0.020398	0.02095	0.020933
6	0.000052	0.000069	0.020846	0.021029	0.02081
7	0.000177	0.000236	0.020977	0.020889	0.020956
8	0.000785	0.000903	0.011611	0.021021	0.021508
9	0.003875	0.00382	0.021122	0.021431	0.02165
10	0.019014	0.019929	0.023768	0.021511	0.021501
11	0.108076	0.084597	0.027506	0.026077	0.023876
12	0.601535	0.391066	0.0896	0.080821	0.0545
13	3.67675	2.22827	0.45447	0.337369	0.267154
14	24.39886	13.664899	2.207753	2.135894	1.340249
15	171.975		14.42721	13.07912	7.470294
16	1283.21		100.8735	95.45986	

160 MPI Processes Trials					
k	Seq	Seq_prune	k_part	k2_part	Prune
1	0.000001	0.000014	0.000014	0.000012	0.000013
2	0.000001	0.000012	0.000098	0.000037	0.000046
3	0.000012	0.000014	0.000163	0.000059	0.000132
4	0.000018	0.00002	0.001174	0.000195	0.004244
5	0.000026	0.000036	0.020434	0.020423	0.020292
6	0.000051	0.000067	0.020404	0.020445	0.020407
7	0.00022	0.000312	0.021007	0.020934	0.021042
8	0.000824	0.000936	0.021092	0.020659	0.020989
9	0.003814	0.005061	0.020921	0.021019	0.020974
10	0.018578	0.015726	0.021081	0.020994	0.021107
11	0.100316	0.078534	0.02434	0.021537	0.021594
12	0.595624	0.404426	0.08766	0.084766	0.058828
13	3.678095	2.22502	0.484696	0.465534	0.281719
14	24.47347	13.58218	3.07399	2.831668	1.593164
15	172.2563		20.35151	19.51661	
16	1284.442		149.269	139.0191	