Final Project
CSC410 - Parallel Computing

Aaron G. Alphonsus

December 11, 2018

1 The N-queens Problem

The goal of n-queens problem is to place n queens on an $n \times n$ chessboard such that no two queens can attack each other. The queen piece in chess can move along rows, columns, and diagonals. This means that any solution we find will not have two queens in the same row, column, or diagonal.

Since no queen can be in the same row or column, we will represent all these solutions as an n-tuple $(x_1, x_2, ..., x_n)$ where column 1 has a queen in row x_1, column 2 has a queen in row x_2, ..., and column n has a queen in row x_n. With this representation, there are $n!$ possible solutions. For each of these solutions, all we need to check is if there are two queens that are on the same diagonal.

2 Benchmarking

In order to benchmark our program, we use the `MPI_Wtime()` function. Along with it, we use the `MPI_Barrier()` function for barrier synchronization so that we are sure that we start the time when all the processes are ready to begin execution and stop the time when they all return. We place these functions before for loop that evaluates each permutation, and after the `MPI_Reduce()` call. Figure 1 is a graph of execution time vs size of the problem. We used 65 processes for this. The execution time increases 'factorially' since for each increase in n by 1, The number of solutions to evaluate is $n!$. Initially there isn't much change in total execution time as the problem execution time is smaller than the overhead. Once it gets to larger numbers, the program execution time dwarfs the overhead time.

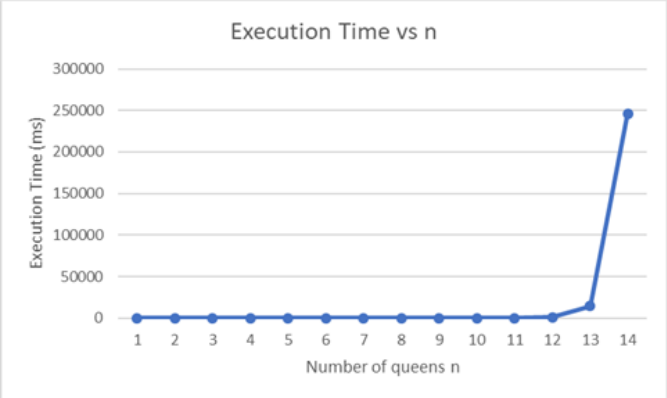


Figure 1: Execution time as problem size increases

3 Performance Analysis

In order to do the performance analysis, we used the similar methods in our benchmarking step but this time we kept the size of the problem fixed, and varied the number of processes. Figure 2 shows the decrease in execution time with the increase in number of processes.

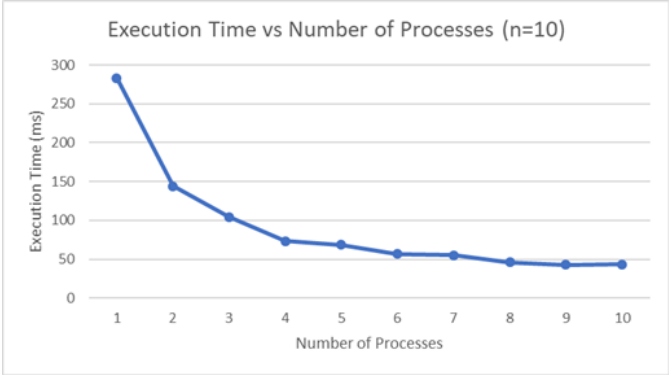


Figure 2: Execution time as number of processes increase

In our performance analysis we also calculated the speedup, efficiency, Karp-Flatt Metric, and isoefficiency relation. Using the experimentally determined serial fraction f_e from the Karp-Flatt Metric, we calculate the scaled-speedup. Amdahl's Law yields the same value as speedup when we use f_e in our calculation.

3.1 Speedup and Efficiency

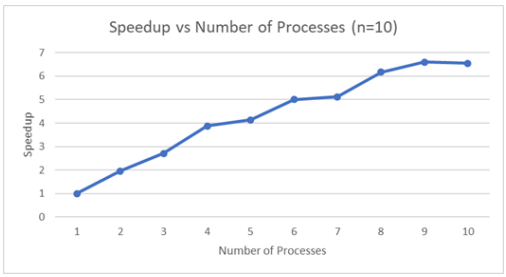
The speedup $\psi(n, p)$ is the ratio between sequential execution time and parallel execution time.

$$\psi(n, p) = \frac{T(n, 1)}{T(n, p)}$$

Efficiency $\epsilon(n, p)$ is defined as the speedup divided by the number of processors used:

$$\epsilon(n, p) = \frac{\psi(n, p)}{p}$$

The speedup and efficiency for our program as a function of increasing processes can be seen in figure 3. You can see that we get steady speedup initially which starts flattening out towards the end. However, the more processes we add, the less we utilize them as the efficiency goes down steadily with increasing numbers of processes



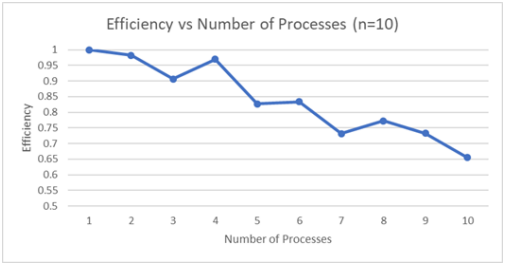


Figure 3: Speedup and Efficiency vs Number of processes

3.2 The Karp-Flatt Metric

The Karp-Flatt metric provides a means to use empirical data to learn something about parallel overhead as well as the amount of inherently sequential computation in a parallel program. The basic idea is to collect data from several runs of the program with increasing numbers of processes.

Definition: Given a parallel computation showing a speedup $\psi(n,p)$ on p processes, where $p > 1$, the experimentally determined serial fraction f_e is defined to be:

$$f_e = \frac{1/\psi(n,p) - 1/p}{1 - 1/p}$$

p	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
$\psi(n,p)$	1.96	2.72	3.88	4.13	5.00	5.12	6.18	6.60	6.55
f_e	0.018	0.052	0.011	0.052	0.040	0.061	0.042	0.046	0.058

What we can see here is that the serial fraction is trending upwards as the number of processes increase. This indicates that the main reason for the poor speedup is parallel overhead. This could be time spent in process startup, communication, or synchronization, or it could be an architectural constraint.

3.3 Scaled Speedup and Isoefficiency Relation

Definition: Given a parallel program solving a problem of size n using p processors, The maximum speedup ψ achievable by this program is:

$$\psi \leq p + (1-p) \cdot s$$

where s is the fraction of execution time spent in serial code. We use the experimentally determined value for the serial fraction from Karp-Flatt in our calculation of the scaled speedup. Figure 4 shows scaled speedup and speedup plotted as a function of increasing processes.

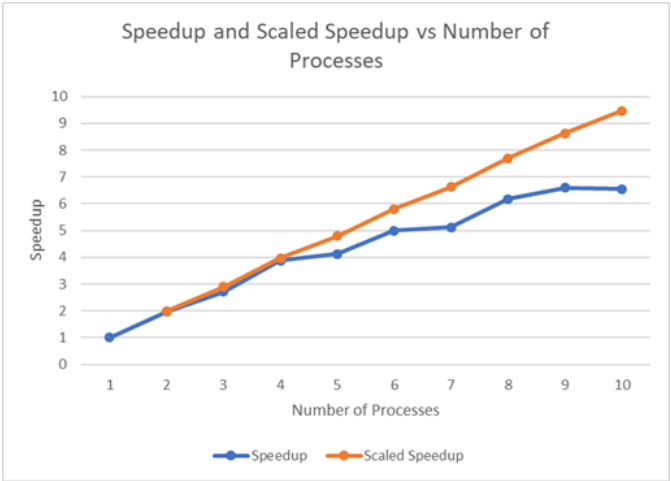


Figure 4: Execution time as number of processes increase

We show the isoefficiency relation for our program graphically in figure 5. Notice that C is a lower bound for the fraction $T(n,p)/T_0(n,p)$. The isoefficiency relation is the following inequality:

$$T(n,1) \geq C \cdot T_0(n,p)$$

where T(n, 1) denotes the sequential time, $T_0(n,p)$ denotes parallel overhead and $C = \epsilon(n,p)/(1-\epsilon(n,p))$

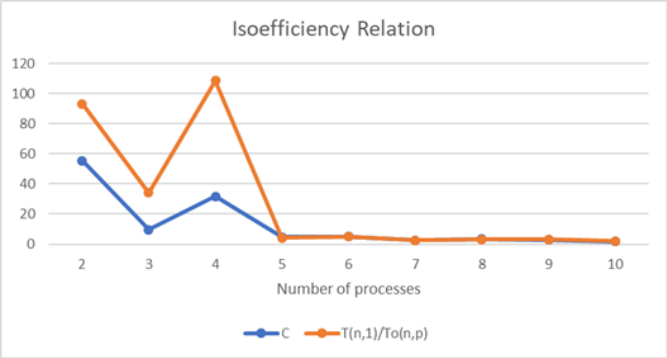


Figure 5: Execution time as number of processes increase

4 Algorithms and Libraries

To solve this problem we used the Task/Channel model to design a parallel algorithm. We began with a cyclic mapping, where the index i of the $n!$ possible solutions were given to each process one-by-one. The process would calculate the i^{th} permutation of the array $(0, 1, ..., n)$, and then test to see if the permutation was a valid solution to place the n queens. We knew calculating the i^{th} permutation every time couldn't be efficient, so we decided to agglomerate the tasks.

With this new mapping, each process is given a contiguous chunk of permutations to evaluate. This is useful because the `std::next_permutation()` function in the C++ `<algorithm>` library is more efficient than calculating the i^{th} permutation each time. The first permutation is calculated using the `ithPermutation()` function that we wrote, but `std::next_permutation()` is used for subsequent permutations.

Each permutation is then checked to see if it is a valid solution by going through every pair of queens and seeing if their row difference is equal to their column difference. This would indicate that they are on the same diagonal. Every process keeps a count of the number of solutions locally. In order to get the count of all the solutions, we use the reduction operation `MPI_Reduce()` to sum up the local sums of each process.

4.1 Libraries used

- `<algorithm>`
- `<math.h>`
- `<mpi.h>`
- `<stdio.h>`
- `<stdlib.h>`
- `<vector>`

5 Functions and Program Structure

The program has 4 functions:

- `main`
- `usage`

- `ithPermutation`
- `checkDiag`

5.1 `main`

Arguments:

- `int argc`: Number of command-line arguments.
- `char* argv[]`: Pointer array storing each command-line argument.

Returns: `0` indicating normal termination.

Description:

- Initializes MPI library, calculates process ranks and total number of processes.
- Allocates contiguous chunks of permutation ranges to be checked by each process.
- Calculates permutations and calls the solution validation function `checkDiag()`.
- Keeps track of the number of local solutions. Performs a reduction to calculate total number of solutions.
- Prints out number of solutions and execution time. Prints out solutions if print flag set.

5.2 `usage`

Arguments:

- `char* prog_name`: Character array containing the name of the program.

Returns: void

Description: Prints a message explaining how to run the program.

5.3 `ithPermutation`

Arguments:

- `int n`: digits to permute
- `unsigned long long i`: index of the permutation.

Returns: `int* perm`: The i^{th} permutation of the array $(0, 1, ..., n)$

Description:

- Calculates the factoradic representation of the decimal number i.
- Uses the factoradic to obtain the i^{th} permutation.

5.4 `checkDiag`

Arguments:

- `int n`: number of queens and size of chessboard.
- `int perm[]`: Possible solution to be accepted or rejected.

Returns: `int`: 1 if no queens share a diagonal, 0 otherwise.

Description: For each pair of queens, check if horizontal distance from each other is the same as vertical distance. If so, the queens are on the same diagonal.

6 Compilation and Usage

Compilation: `mpiCC -g -Wall -o nqueens nqueens.C -lm` OR `make`
Usage: `mpirun -np <number_of_processes> -hostfile <path_to_hostfile> ./nqueens <n> <print>`

The program can be compiled using the command `make`. To get rid of the executable in the folder, run the command `make clean`.

The program takes in an integer n indicating number of queens and size of the chessboard, as well as a 0 or 1 suppressing or allowing printing of solutions to the console window. The program prints out the number of solutions and execution time by default.

7 Testing and Verification

For verification, as we built the program we used small values of `n` (n=1 to n=6) and printed out the solutions. We found these solutions by hand and confirmed that they matched the ones we were getting with our program.

For slightly larger solutions, we sorted our solutions and wrote them out to a list. We compared this list with the output of an n-queens program found online that we edited slightly to give the similar tuple output that we have.

For large values of n and large numbers of solutions, we tested our program based on getting the right number of solutions 6. We decided that the odds that we got the same number of solutions but had a wrong solution for multiple values of n was unlikely :)

```
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 1 0
1
Execution time    0.403 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 2 0
0
Execution time    0.431 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 3 0
0
Execution time    0.570 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 4 0
2
Execution time    2.326 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 5 0
10
Execution time    0.771 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 6 0
4
Execution time    0.505 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 7 0
40
Execution time    0.678 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 8 0
92
Execution time    3.433 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 9 0
352
Execution time    1.736 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 10 0
724
Execution time   11.326 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 11 0
2680
Execution time  114.078 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 12 0
14200
Execution time 1059.758 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 13 0
73712
Execution time 14995.357 ms
7296837@linux102 final >>mpirun -hostfile ~/.openmpi/hostfile ./nqueens 14 0
365596
Execution time 296997.985 ms
```

Figure 6: Number of solutions till n=14

8 Files Submitted

- `final.pdf`
- `makefile`

- `nqueens.C`
- `latex/`