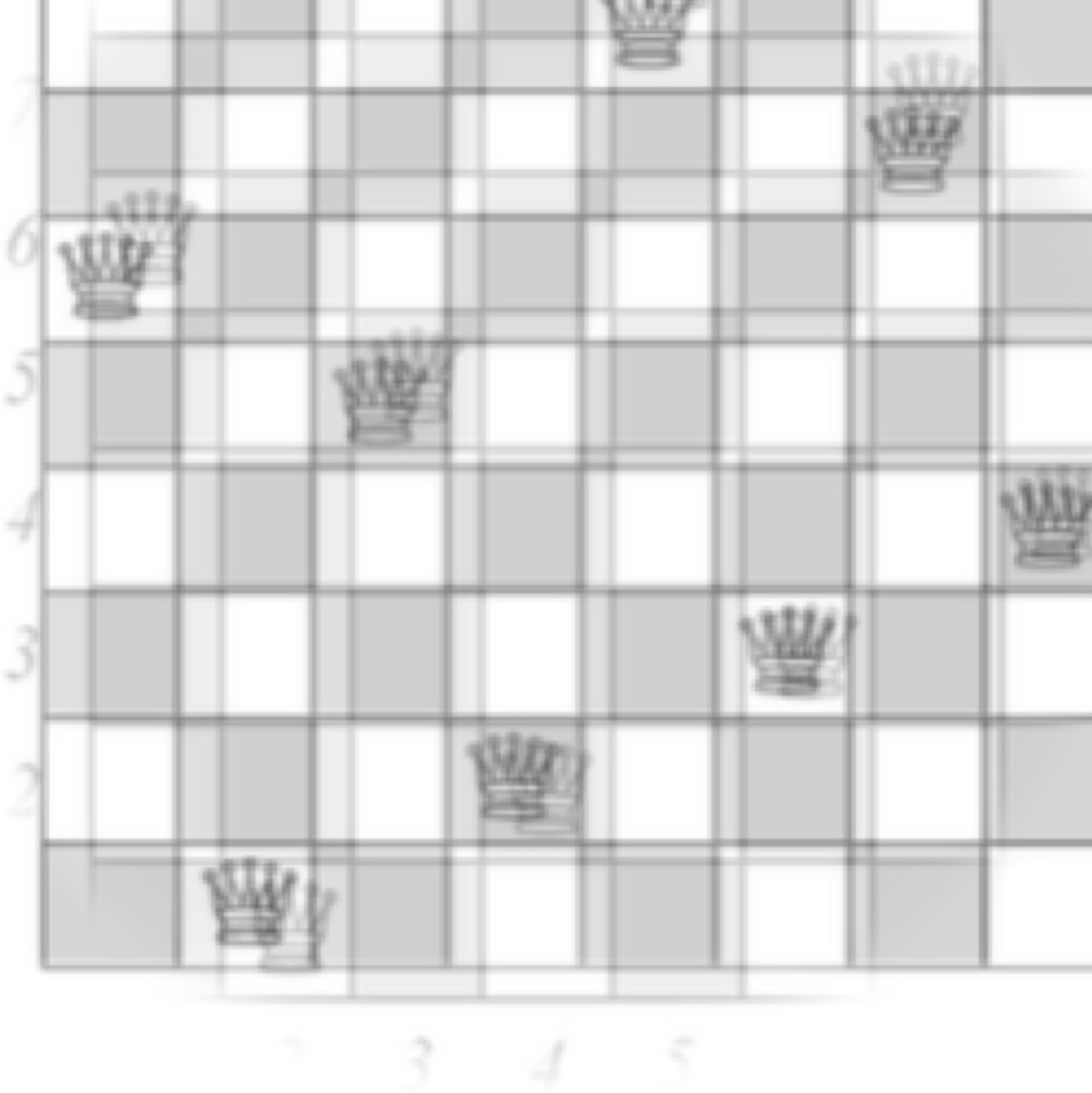


N QUEENS PARALLELIZATION

Lab Work III Parallel & Distributed Computing

Javier Garcia Santana

Table of Contents

1. Problem Description

The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.

2. Parallelization of the algorithm

- Work Division among Threads:

OpenMP's #pragma omp parallel creates a team of threads.

#pragma omp single ensures that the following block is only executed by one thread. Within this single thread region, a loop is used to create tasks for initial queen placements.

#pragma omp task generates tasks for each initial queen placement, distributing the work among the available threads.

- Task Allocation:

The solve function creates tasks for each possible initial placement of the first queen. Each task then explores different paths of queen placements recursively.

- Thread Synchronization:

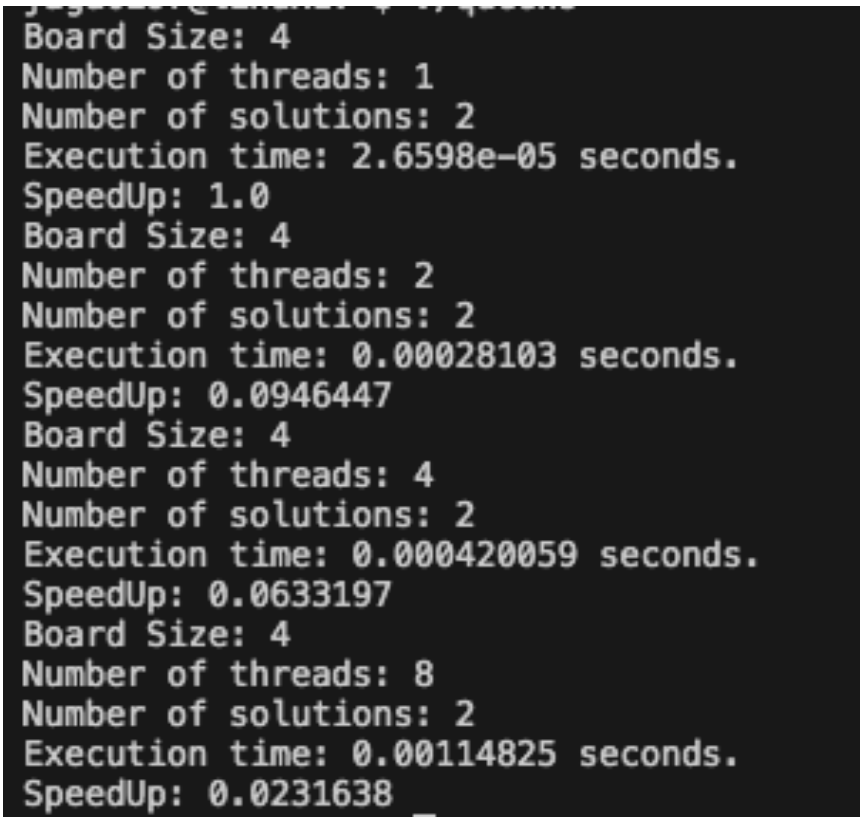
Within the placeQ function, #pragma omp critical is used to ensure mutual exclusion when incrementing the numofSol variable to avoid race conditions. This provides a form of thread synchronization, ensuring only one thread at a time can increment the solution count.

3. Experiment

The experiment was conducted in the MIF cluster, giving us access to 12 cores, which is plenty for the tests since only 8 cores are needed.

When experimenting with different table sizes, the speedup is much greater when we have more than 8 queens. This is not the desired outcome, since with parallelization what we want is to optimize the algorithm for every possible case. However, because of this problem's nature, it is not feasible.

4x4 Size table:



8x8 Size table:

```
Board Size: 8
Number of threads: 1
Number of solutions: 92
Execution time: 0.000872194 seconds.
SpeedUp: 1.0
Board Size: 8
Number of threads: 2
Number of solutions: 92
Execution time: 0.000591552 seconds.
SpeedUp: 1.47442
Board Size: 8
Number of threads: 4
Number of solutions: 92
Execution time: 0.000661709 seconds.
SpeedUp: 1.31809
Board Size: 8
Number of threads: 8
Number of solutions: 92
Execution time: 0.00193235 seconds.
SpeedUp: 0.451365
```

12x12 Size table:

```
Board Size: 12
Number of threads: 1
Number of solutions: 14200
Execution time: 0.401196 seconds.
SpeedUp: 1.0
Board Size: 12
Number of threads: 2
Number of solutions: 14200
Execution time: 0.212351 seconds.
SpeedUp: 1.88931
Board Size: 12
Number of threads: 4
Number of solutions: 14200
Execution time: 0.115208 seconds.
SpeedUp: 3.48236
Board Size: 12
Number of threads: 8
Number of solutions: 14200
Execution time: 0.0728884 seconds.
SpeedUp: 5.50425
```

This can be due to a certain number of reasons:

Problem's Size: For smaller board sizes, the problem might be too small to fully benefit from parallelization. With only 4 or 8 queens, the search space is relatively small, and the overhead of thread creation and management in parallel processing could outweigh the benefits gained from parallel execution.

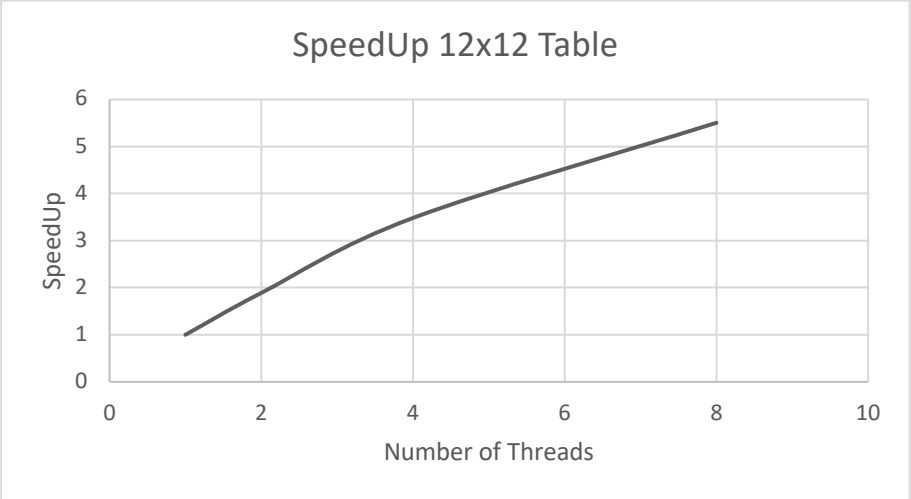
Overhead: Parallel processing comes with some overhead, such as thread creation, synchronization, and management; which might not be justified for very small problem sizes.

Parallel Overheads: The overhead associated with parallel processing could be more noticeable with a small problem size, impacting the speedup. Threads need to be created, synchronized, and managed, and these operations might take more time than the actual computation for smaller instances of the N-Queens problem.

Limited Work Distribution: With a small problem size, the work distribution among threads might not be efficient. Each thread might finish its small task quickly, leading to waiting time as the threads need to synchronize before completing the whole task.

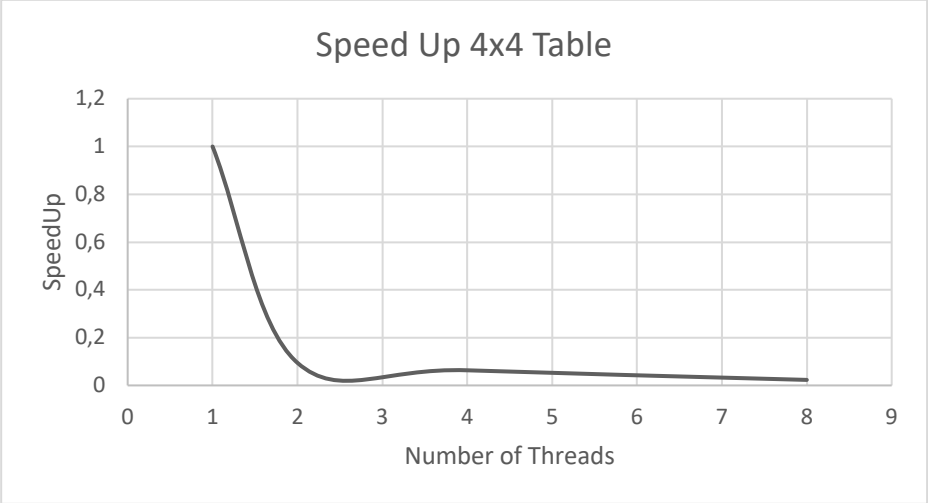
4. Conclusions

The maximum speed up achieved is, as I showed earlier in the experiment's outcome of about x5.50. This is best case scenario, using a table size bigger than the most significant number of threads, 8. This increase in speed is not quite linear but almost. The reduction in time is every time, slightly less, since we have to take into account certain things that affect the result when testing parallelization in the real world.



Firstly, there is **Amdahl's law**, which states that the overall speed-up of a program is limited by the sequential portion of the code. Even with a perfectly parallelizable program, the sequential part constrains the achievable speed-up. So, because of this we have the first constrain. The time it takes to execute the sequential part of the code is always present, hence, making a severe impact in the linear growth.

Secondly, we have some problems related to thread creation and communication, which also take time, limiting the speed up. This is one of the major issues I encountered when experimenting with a small size table and a higher count of threads as it shows on the following graph:



Since the complexity of the problem is very basic (has 2 solutions) it takes less time for a single thread to resolve it than to create and synchronize 2 or more threads to find the to solution. So, for very basic problems, parallelization is not an efficient way of improving an algorithm's execution time.

Lastly, cache and hardware limitations are also a concern. As the number of threads increases, the effectiveness of cache usage might decrease due to increased contention for cache resources. Hardware limitations of the system, like the number of physical cores or memory bandwidth, can also restrict linear scaling. I do not think this is one of the main culprits to not having a linear growth but it is still a factor to have into consideration.

In summary, the ideal linear speed-up assumes perfect parallelization without any overhead, contention, or limitations. Real-world factors like those mentioned above often prevent achieving this ideal linear relationship between the number of threads and speed-up. On the other hand, this constrains that are preventing us to have a linear increase in speed up, do not limit the scalability of the algorithm. The bigger the table size and the bigger the thread count, the more decrease in time we see, making the constrains mentioned before almost unnoticeable.