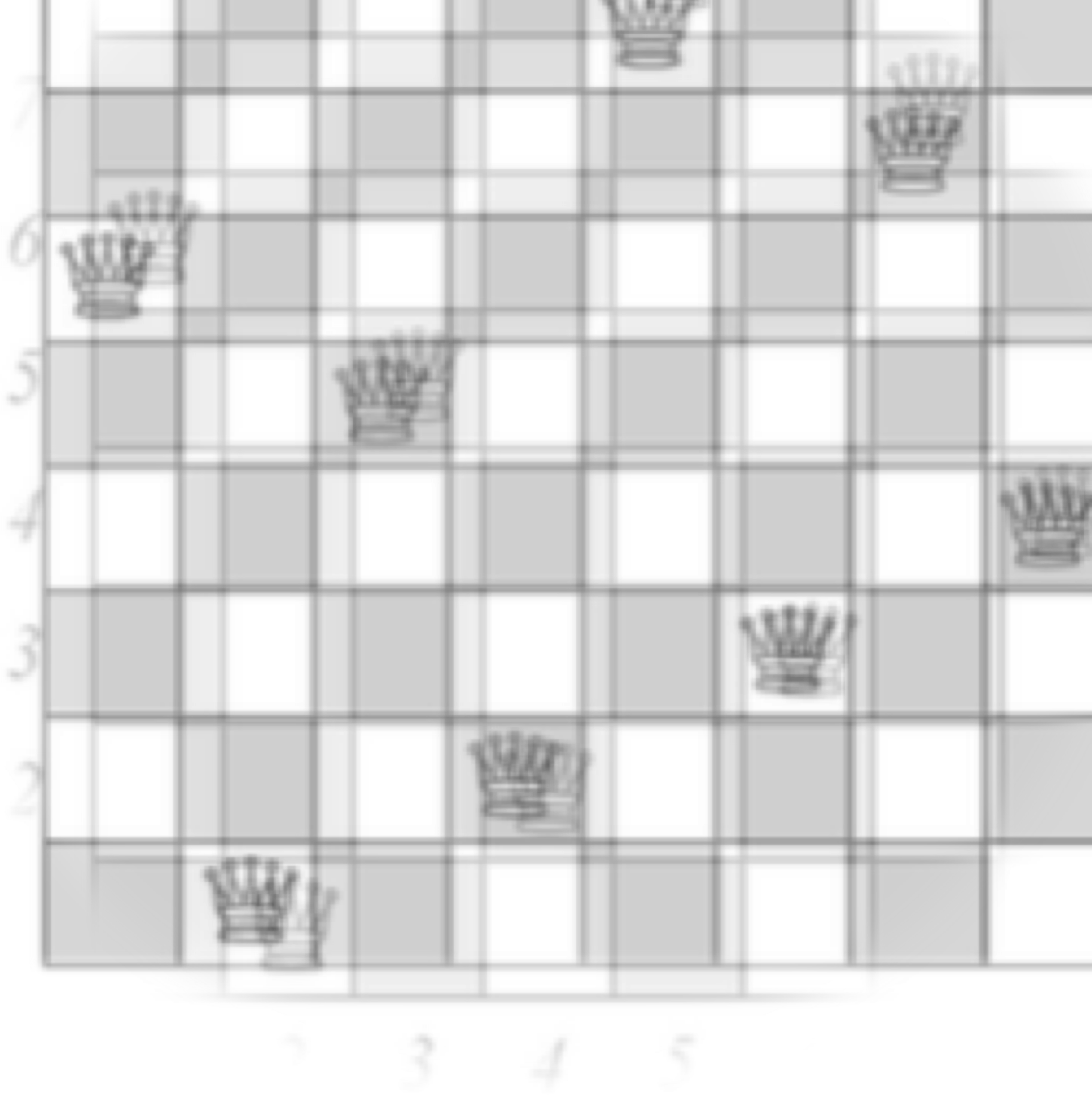


N QUEENS PARALLELIZATION

Lab Work IV 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

N-Queens problem is parallelized by dividing the chessboard among MPI processes, with each process independently finding solutions for its assigned portion. The program then uses MPI reduction to collect and sum up the local solution counts, providing a global count of solutions. The use of MPI allows for distributed memory parallelism across multiple processes.

- **MPI Initialization**

  MPI is initialized using MPI_Init(&argc, &argv).

  The rank of the current process (rank) and the total number of processes (size) are obtained using MPI_Comm_rank and MPI_Comm_size.

- **Command-Line Arguments**

  The program expects two command-line arguments: <board_size> and <num_threads>.

  <board_size> represents the size of the chessboard (number of rows and columns).

  <num_threads> is used as a command-line parameter but might be intended to specify the number of MPI processes rather than threads.

- **Partitioning the Problem**

  The N-Queens problem is divided among MPI processes. Each process is assigned a range of rows on the chessboard.

  The program calculates the start and end rows assigned to each process based on the total number of processes (size) and the board size (boardSize).

```
int rowsPerProcess = boardSize / size;
int startRow = rank * rowsPerProcess;
int endRow = (rank == size - 1) ? boardSize : (rank + 1) *
rowsPerProcess;
```

- **Parallelized Solution Generation**
  The function placeQueens is responsible for generating solutions for the assigned portion of the chessboard.

  Inside this function, a vector *'queens'* is initialized for each process, and the solveNQueens function is called to find solutions for the specified rows.

- **Communication and Coordination**

  MPI processes operate independently on their assigned portion of the problem.

  There is no explicit communication between processes during the solution generation phase. Each process works on its subset of rows.

- **Solution Counting and Reduction**

  Each process independently counts the number of solutions (localSolutions) for its assigned portion.

  The MPI_Reduce function is then used to sum up the local solution counts from all processes, and the result is stored in globalSolutions.

  This reduction operation is performed on the master process (rank 0).

```
int localSolutions = 0;
placeQueens(boardSize, startRow, endRow, localSolutions);

int globalSolutions;
MPI_Reduce(&localSolutions, &globalSolutions, 1, MPI_INT,
MPI_SUM, MASTER_RANK, MPI_COMM_WORLD);
```

- **Timing**

  The execution time of the parallelized solution generation is measured using MPI_Wtime().

- **Output**

  The master process (rank 0) outputs information about the board size, the number of processes, the number of solutions, and the execution time.

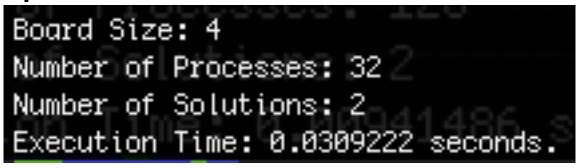
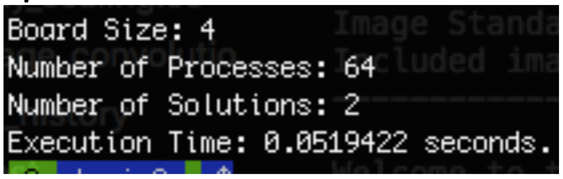
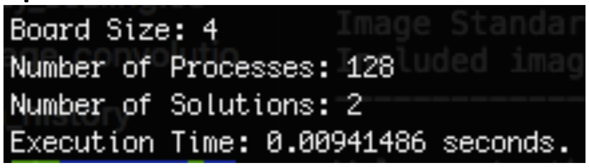
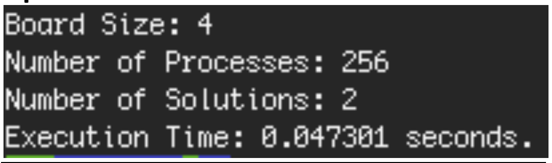
- **MPI Finalization**

  The program concludes with MPI finalization using MPI_Finalize().

3. Experiment

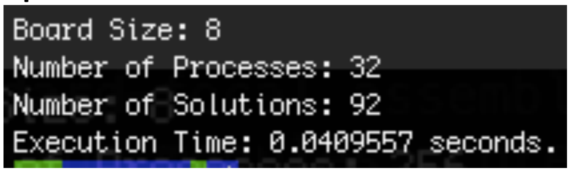
The experiment was conducted in the MIF hpc cluster. Giving us the ability to test up to 256 processes

The experiment sample will be the same as with OpenMP. However, I will add more cases for using different processes. The number of said processes will go from 64 to 256.
Previously, 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, now we have another variable to play with, but it won't affect the speed up drastically in those small table cases.

4x4 Size table:
Np=32

```
Board Size: 4
Number of Processes: 32
Number of Solutions: 2
Execution Time: 0.0309222 seconds.
```
Np =64

```
Board Size: 4
Number of Processes: 64
Number of Solutions: 2
Execution Time: 0.0519422 seconds.
```
Np =128

```
Board Size: 4
Number of Processes: 128
Number of Solutions: 2
Execution Time: 0.00941486 seconds.
```
Np =256

```
Board Size: 4
Number of Processes: 256
Number of Solutions: 2
Execution Time: 0.047301 seconds.
```

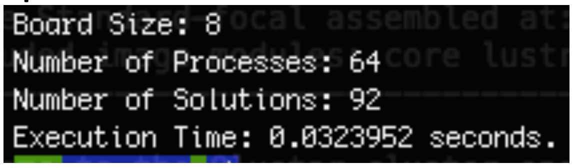
8x8 Size table:

Np=32



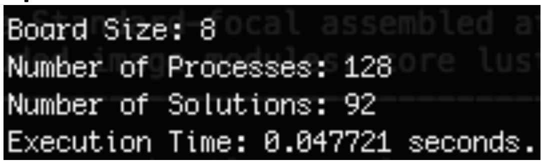
```
Board Size: 8
Number of Processes: 32
Number of Solutions: 92
Execution Time: 0.0409557 seconds.
```

Np=64



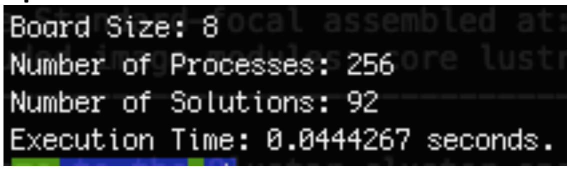
```
Board Size: 8
Number of Processes: 64
Number of Solutions: 92
Execution Time: 0.0323952 seconds.
```

Np=128



```
Board Size: 8
Number of Processes: 128
Number of Solutions: 92
Execution Time: 0.047721 seconds.
```

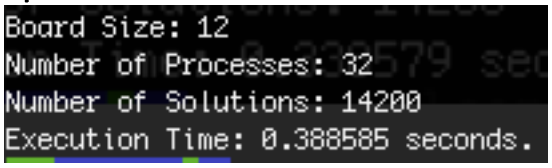
Np=256



```
Board Size: 8
Number of Processes: 256
Number of Solutions: 92
Execution Time: 0.0444267 seconds.
```

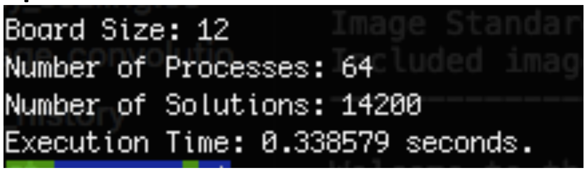
12x12 Size table:

Np=32



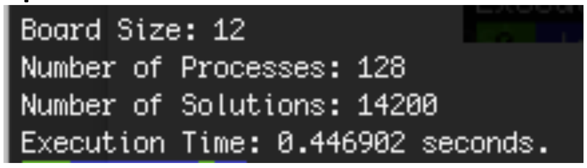
```
Board Size: 12
Number of Processes: 32
Number of Solutions: 14200
Execution Time: 0.388585 seconds.
```

Np=64



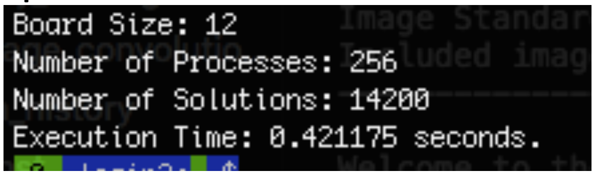
```
Board Size: 12
Number of Processes: 64
Number of Solutions: 14200
Execution Time: 0.338579 seconds.
```

Np=128



```
Board Size: 12
Number of Processes: 128
Number of Solutions: 14200
Execution Time: 0.446902 seconds.
```

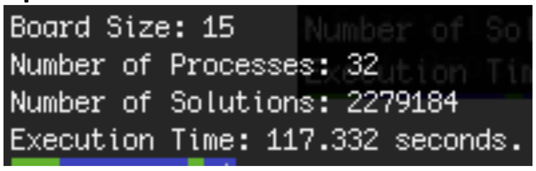
Np=256



```
Board Size: 12
Number of Processes: 256
Number of Solutions: 14200
Execution Time: 0.421175 seconds.
```

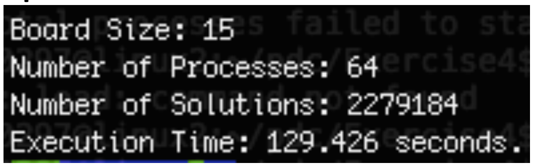
15x15 Size table:

Np=32



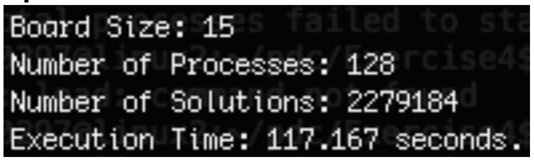
```
Board Size: 15
Number of Processes: 32
Number of Solutions: 2279184
Execution Time: 117.332 seconds.
```

Np=64



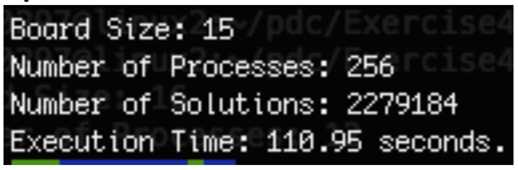
```
Board Size: 15
Number of Processes: 64
Number of Solutions: 2279184
Execution Time: 129.426 seconds.
```

Np=128



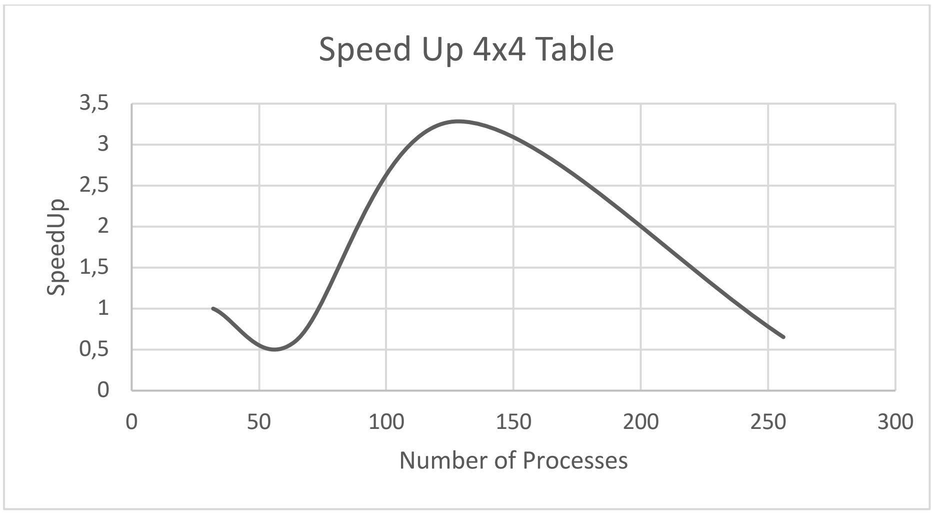
```
Board Size: 15
Number of Processes: 128
Number of Solutions: 2279184
Execution Time: 117.167 seconds.
```

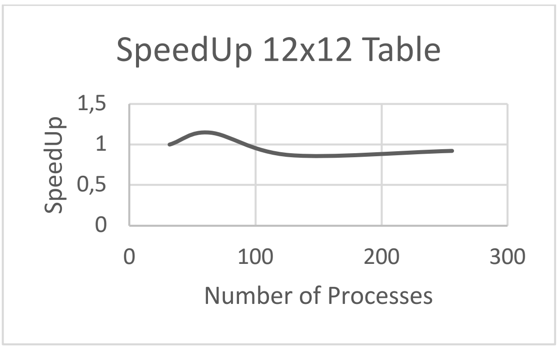
Np=256

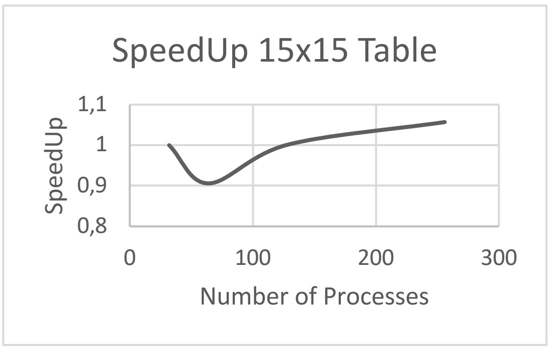


```
Board Size: 15
Number of Processes: 256
Number of Solutions: 2279184
Execution Time: 110.95 seconds.
```

4. Conclusions







The relationship between the number of processes in MPI and the speedup of a parallel program is not always straightforward. The impact of increasing the number of processes on speedup depends on several factors, including the nature of the parallel algorithm, the characteristics of the problem being solved, and the underlying hardware.

In theory, increasing the number of processes can lead to improved parallelism and potentially higher speedup. However, there are practical considerations and limitations to be aware of:

Communication Overhead:

As the number of processes increases, the amount of communication between processes may also increase. Excessive communication can lead to increased overhead and may offset the benefits of parallelism.

Load Balancing:

If the workload is not evenly distributed among processes, some processes may finish their work faster than others, leading to idle time for certain processes. Load balancing becomes crucial for achieving optimal performance.

Parallel Efficiency:

The efficiency of parallelization, often measured by parallel efficiency or scalability, can be affected. If the parallel algorithm is not well-suited for a large number of processes, the efficiency may decrease.

Amdahl's Law:

Amdahl's Law describes the limit to the speedup achievable by parallelization. If there are serial portions of the code that cannot be parallelized, the overall speedup will be limited.

To determine the optimal number of processes for a specific problem and algorithm, it's necessary to perform experiments and benchmark the program with different configurations. This process, known as performance tuning, involves adjusting parameters such as the number of processes to find the configuration that maximizes performance.

In summary, while increasing the number of processes has the potential to improve speedup, it is not a guarantee. Careful consideration of communication overhead, load balancing, and the characteristics of the parallel algorithm is essential to achieve optimal performance. Moreover, it is important to clarify that if we use more processes than number of rows there are, some processes may have very little work to do, leading to inefficient use of resources. On the other hand, if there are too few processes, the workload may not be adequately parallelized. Moreover, as we have seen in the graphs above, the nature of this problem only benefits from a big number of processes when we have millions of possible solutions or when the number of processes can divide the workload more efficiently(4x4 table).