|  |
| --- |
| Edinburgh Napier University |
| SET10108 – Concurrent and Parallel Systems (Coursework 2) |
| Michael Suttie - 40541559 |

|  |
| --- |
| Suttie, Michael  12-6-2022 |

Table of Contents

[1. Introduction 2](#_Toc121251589)

[1.1. Hardware Specs 2](#_Toc121251590)

[2. Serial 3](#_Toc121251591)

[2.1. Recursion Problem 3](#_Toc121251592)

[2.2. New Solution 3](#_Toc121251593)

[2.3. Serial Performance Results 3](#_Toc121251594)

[3. Parallel – OpenMP 4](#_Toc121251595)

[3.1. Method of Parallelisation 4](#_Toc121251596)

[3.2. OpenMP Performance Results 4](#_Toc121251597)

[4. Parallel – GPU (CUDA) 5](#_Toc121251598)

[4.1. Method of Parallelisation 5](#_Toc121251599)

[4.2. OpenMP Performance Results 5](#_Toc121251600)

[5. Comparison 6](#_Toc121251601)

[6. Conclusions 7](#_Toc121251602)

Table of Figures

[Figure 2‑1: Serial Implementation 4](#_Toc121435124)

[Figure 2‑2: Serial N-Queens Performance Table 5](#_Toc121435125)

# Introduction

The purpose of this coursework is to solve the N-Queens problem – the problem of placing chess queens on an N x N chessboard in such a way that no two queens are a threat to another. This means that no queen can share a row, column or a diagonal space. Given the implementation provided, the task is to modify the existing serial program so that it is non-recursive and then to implement the non-recursive solution in parallel. It is important to ensure that the recursive elements are removed, as parallelisation through OpenMP and using the GPU does not play will with recursive code.

The parallelisation should be done twice; once in OpenMP and then again using the GPU using CUDA or OpenCL, and should run for N values between 4 and 10 – with the option to increase the threshold should the Laptop have the ability to handle it.

## Hardware Specs

The machine used for the creation of this program as well as running the experiments are an ASUS TUF F15 with the following specs:

* CPU – 11th Gen Intel(R) Core(TM) I5 11400H @ 2.70GHz (6 cores, 12 logical processors)
* RAM – 16GB DDR4 3200mhz
* GPU – NVIDIA GeForce RTX 3050 Ti Laptop GPU (4GB)
* OS – Windows 11 Home (Build 22000.1098)

# Serial

## Implementation

The initial serial program provided for the coursework completes the task using recursion. The code for this can be seen below:

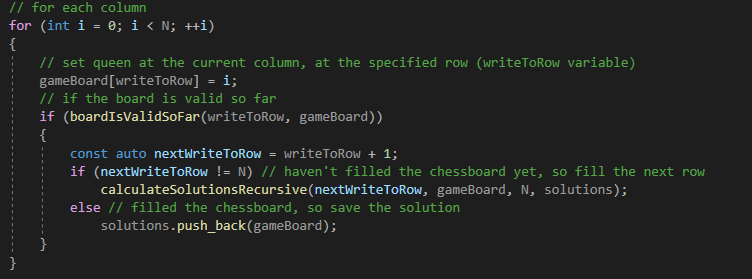


Figure 2‑1: Serial Implementation

Essentially, if the chess board is deemed valid, the function calculateSolutionsRecursive() will call upon itself continually in an attempt so solve the rest of the problem and send the solutions back to the solutions vector.

This solution is fine when done serially, however, when implemented in parallel, this won’t be the case. The reason for this is basically in the name, as recursion often relies on the results of previous recursive calls and typically means that the order of the recursive calls are important to the operation of the algorithm. For this reason, the future two sections on OpenMP and CUDA will run a different algorithm that aims to reduce or remove recursion completely.

## Serial Performance Results

This section will showcase and discuss the results specific to the running of the original recursive code provided. These results will be used as a comparison in the future sections when discussing the parallelisation of the N-Queens problem.

The performance of the serial program was measured by running it 10 times and recording how long each N-Queens problem took to solve. The table below shows these recorded times and their averages. The execution was so quick that the std::chrono couldn’t register a time at all for N = 4 through to N = 8, with a slight outlier in run 3 for N = 8. The only time properly registered was when calculating the solutions for N = 9 and N = 10. Even at that, the times were miniscule.

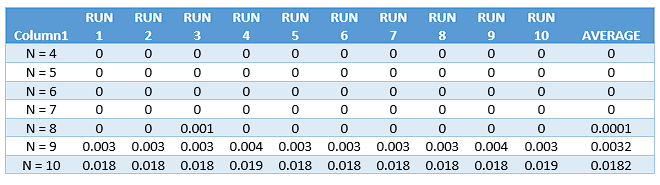


Figure 2‑2: Serial N-Queens Performance Table

The averages of these runs were then taken and the trend seems to show that the times get worse as the complexity of the problem increases, which is to be expected given the calculations that are occurring – a trend that will likely be replicated in the analysis of both parallel implementations too.

Figure 2‑3: Serial N-Queens Average Times

# Parallel – OpenMP

## Removing Recursion

As was touched upon in the Serial section – the first hurdle to overcome was changing the N-Queens solution from a recursive solution to a non-recursive solution. This was initially attempted using a backtracking algorithm. Essentially, backtracking works by trying to incrementally build the solution piece-by-piece and then going back on itself and undoing previous steps if the algorithm decides that it cannot go any further (Datta, 2022). The implementation of this can be seen in the figure below, inspiration of how to implement this was taken from Oxford College of Emory (Oxford College, n.d.)

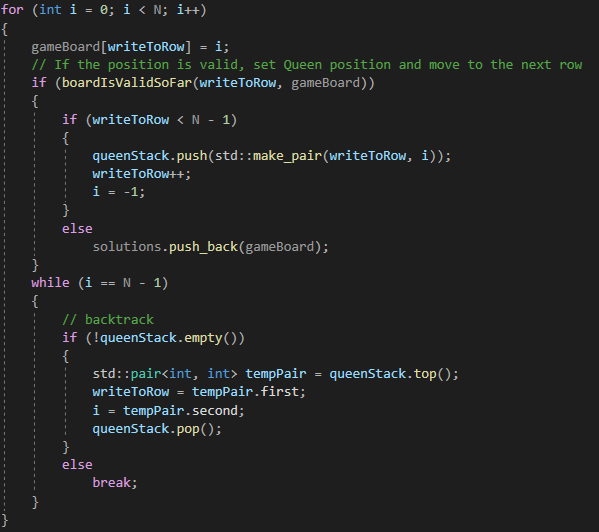
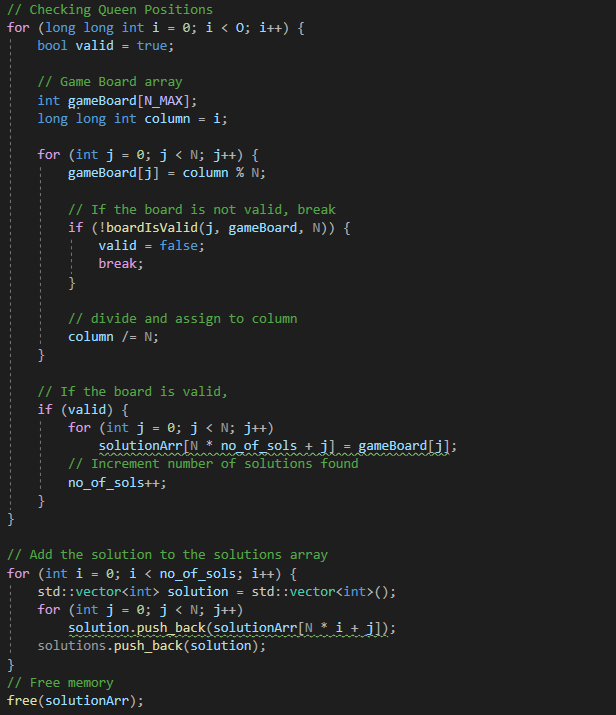


Figure 3‑1: Initial Non-Recursive Attempt

This code, however, proved to be difficult to implement in a way that would be parallelisable as it typically relies on recursion to be effective. Therefore, the decision was made to look into another iterative approach; the brute-force approach. It would be simple, and therefore likely to be rather slow, however it would be easier to parallelise using OpenMP and CUDA.

An acceptable approach to N-Queens with brute force was pieced together using techniques and examples found across the internet – these sources can be found in the references section of this paper and will be referenced in the following figure.

This brute force implementation will check the possible positioning of the all queens on the board and determine whether or not the positioning is valid. If the board is invalid, then the valid Boolean will be set to false. If the board remains valid, then the solution can be added to the solution array and the number of solutions can be incremented. A new solution vector is then created and populated with the solutions in the solution array and pushed back into the solution vector before the memory is freed up.



Initially, the game board was a vector – this came with some slowdown and as such was changed to an array. From working with OpenMP in the past, it was known that the array would likely work better across multiple threads than the vector would have anyway, so this change was made in preparation for the parallelisation.

The resources used to arrive at this solution are listed below:

* Simran (Simran, 2021)
* Stack Exchange (Stack Exchange, 2019)
* Geeks For Geeks (Geeks for Geeks, 2022)

Figure 3‑2: Final Brute Force Implementation

To accommodate the changes in the solution calculations, some changes had to be made in the boardIsValid method, however, these changes were easy to implement in comparison to researching and implementing the brute force method, especially as the brute force method was designed with the boardIsValids already functioning method in mind.

With this now in place, it came time to parallelise the new brute force solution using OpenMP.

## Method of Parallelisation

## OpenMP Performance Results

# Parallel – GPU (CUDA)

## Method of Parallelisation

## OpenMP Performance Results

# Comparison

# Conclusions