Parallel Counting for the N-Queens Problem

Ben Howard
1905021

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Department of Computer Science
Swansea University

March 23, 2023

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed [illegible] (candidate)

Date 02/05/2022

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed [illegible] (candidate)

Date 02/05/2022

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed [illegible] (candidate)

Date 02/05/2022

Abstract

The N-Queens problem is a classic computational problem involving the placement of *n* queens on a *n* size chess board in which no two queens can attack one another. There are numerous approaches to solve the problem in full by calculating every possible solution for a given board size. In this paper I will attempt to solve this problem using parallel computing and researching various algorithms that can be implemented. I implement two backtracking algorithms one of which makes use of arrays to represent the queen locations and the other uses bit patterns. Tests are automatically run that gather the run time of the algorithm as well as if it accurately produced the correct value. I test the program on Supercomputing Wales HPC on a large cluster of processor cores.

Acknowledgements

I would like to thank my project supervisor Oliver Kullmann for his support and guidance. I would also like to thank Supercomputing Wales for allowing me to use their hardware as well as Thomas Pritchard for assisting me in fixing the initial issues I had with the HPC.

Contents

List of Tables v

List of Figures vi

1 Introduction 1

1.1 Aims and Objectives 2

2 Background 3

2.1 N-Queens 3

2.2 Related Work 6

2.3 Parallelisation 7

2.4 Bit Patterns 9

3 Project Management 13

3.1 Time Management 13

3.2 Risk Mitigation 14

3.3 Testing 15

4 Implementation 16

4.1 N-Queens in Java 16

4.2 Serial N-Queens in C++ 16

4.3 Parallel N-Queens on my PC 17

4.4 Parallel N-Queens on a HPC 18

5 Results 20

5.1 PC Backtracking Results 21

5.2 HPC Results 23

5.3 Bitwise Results 24

6 Conclusions and Future Work 26

6.1 Future Work 27

List of Tables

2.1 Total number of solutions for given *n* size. 4

2.2 Time to calculate the total number of solutions using Jeff Somers' program *n* size. 12

5.1 Time to calculate the total number of solutions using arrays and bit patterns. 24

5.2 Time to calculate the total number of solutions using bit patterns and Jeff Somers' implementation. 25

5.3 Fastest time to calculate the total number of solutions using bit patterns 25

List of Figures

2.1 A N-Queens backtracking algorithm shown to get one solution for $n = 2$ 5
2.2 Symmetry between two 4×4 N-Queen solutions. 6
2.3 A solution the 8-Queens problem represented as a 2D boolean matrix 9
2.4 A solution the 8-Queens problem represented as an integer array 9
2.5 Visualisation of the available queen placements using bit patterns. 10
2.6 Converting the column, left diagonal and right diagonal queen locations to available queen placements 10
2.7 Calculating the least significant bit of x 11
2.8 Calculating the next row diagonal queen placements from the previous row 11

3.1 Gantt chart with labels 14

5.1 Run time of the backtracking N-Queens program plotted against the value of N run on PC . 22
5.2 Run time of the program plotted against depth of initial tree to solve the 14-Queens problem 22
5.3 Run time of the backtracking N-Queens program plotted against the value of N run on HPC 23
5.4 Run time of the backtracking 16-Queens program plotted against the number of processors used 24

Chapter 1

Introduction

The N-Queens problem initially started out as the Eight Queens puzzle. First proposed by chess composer Max Bezzel in 1848 to renowned chess players all over Germany [1]. The aim of the puzzle was to place eight queens on the standard eight by eight chessboard. The queen can move vertically, horizontally and diagonally in any direction on the board and thus is the most valuable chess piece on the board excluding the king. It's troublesome to find a single solution to the problem, but the real goal is to find every possible solution. The Eight Queens puzzle was fully solved in 1850 by Franz Nauck who successfully found all 92 solutions [2]. With the succession of this solution, many chess composers began to theorise about non-standard chess boards.

With larger sizes of boards looking to be solved, the problem transformed into the N-Queens puzzle. The rules are identical to the eight queens puzzle although the number of queens to place scales equally with the size of the board. As mathematicians and chess players began to solve for larger board sizes, they found that the number of solutions would increase dramatically. There exists solutions for all natural numbers except for $n = 2$ and $n = 3$ [3]. As in the eight queens puzzle, finding a single non-trivial solution isn't the real goal, but finding all possible solutions for a certain board size.

Interest in computing this problem was popularised by Dijkstra in 1971 when he included it in his paper "A short introduction to the art of programming" [4]. It became a common exercise for backtracking in many artificial intelligence and constraint programming textbooks. Backtracking can provide a complete solution to the N-Queens problem with all solutions found for a given board size. As well as being an exercise in artificial intelligence and constraint programming, the N-Queens problem also lends itself to parallel programming.

The N-Queens problem is a perfectly parallel problem which makes it trivial to implement into a parallel system. This is because there is very little effort in splitting this problem into multiple parallel tasks. We can achieve this by running a solving algorithm on a partly complete board. The simplest option would be to place a queen in each position of a row or column, this would split the workload into eight distinct tasks that could be solved simultaneously. We can produce more parallel tasks by solving the problem to set depths. The exploration of solving the N-Queens problem through parallelism will be the focus of this paper.

1.1 Aims and Objectives

The increasing importance of parallel computing has never been clearer. With the exponential growth of processing and network speeds, parallel computing has become a necessity. Examples of parallel computing can be seen everywhere in today's society from smartphones with multicore processors, to medical research using parallel systems to simulate protein sequence alignment [5]. It is an industry that will keep growing alongside the computation of data as a whole.

High Performance Computing (HPC) is the ability to process data and perform complex calculations at high speed and is a topic I haven't worked with. With that, the bulk of my project will be spent designing and implementing algorithms to run on parallel systems. I will be able to test my code on my personal computer as it has multiple cores. There are many existing N-Queens algorithms that have been executed on HPC's before, I intend to learn how these work and how they were implemented.

Obviously the ultimate aim for this project and most N-Queen problem papers is to calculate the solution to the 28-Queens problem. Although this is an incredibly unlikely aim for myself due to the lack of time and resources I have available at my disposal. Top500 is a project that aims to maintain HPC statistics and ratings on a bi-yearly basis [6]. As of November 2021 the highest rated HPC is the Japanese supercomputer Fugaku with its peak performance sitting at over a exoflop and some seven million cores [7]. Many of the current projects looking to solve the 28-Queens problem comfortably sit in the Top500 with access to these systems for prolonged periods of time. Also many of these projects have a team of academics looking to optimise every aspect of the operation. With my limited resources and access to high performance computers, alongside my set time frame, this will merely be an introduction into the world of parallel computing and thus my aims will be set accordingly.

Aim 1 – To implement a parallel N-Queens algorithm on a HPC.

Aim 2 – To measure and collect the speed and accuracy of the algorithm.

Aim 3 – Use the speed and accuracy data collected to improve and redesign the algorithm.

Aim 4 – Summarise my findings and compare the various algorithms and methods implemented.

Aim 5 – To correctly calculate the highest value of n and to compare my results to other projects.

These aims were set in order to not only achieve the bare minimum of implementing a parallel algorithm on a HPC but analyse it in order to improve it. The improvement and speed of algorithms is an important part of this project so a large period of time will be dedicated to analysing and improving these algorithms. I understand with my limited resources I will be unable to produce any groundbreaking research but will put emphasis on the learning process throughout this project to help others who have similar goals.

Chapter 2

Background

2.1 N-Queens

Calculating the N-Queens problem has been a long tradition in the computer science field and has been famously used in examples of backtracking [8], constraint satisfaction [9] and permutation generation [10]. Although the N-Queens problem is not a constraint satisfaction problem, it can be naturally formulated as one [11]. Brute force techniques are mostly overlooked with the large number of possible arrangements and the low number of possible solutions. Although, as there are set constraints on where the queen can be placed, we can use shortcuts in order to reduce the number of possible arrangements. For example limiting a single queen to a row, we reduce the total number of possible arrangements to 8^8. We could then blindly place a queen in each column/row and although this is still a very poor algorithm, it is one of the most efficient brute force algorithms.

n	Fundamental	All
1	1	1
2	0	0
3	0	0
4	1	2
5	2	10
6	1	4
7	6	40
8	12	92
9	46	352
10	92	724
11	341	2,680
12	1,787	14,200
13	9,233	73,712
14	45,752	365,596
15	285,053	2,279,184
16	1,846,955	14,772,512
17	11,977,939	95,815,104
18	83,263,591	666,090,624
19	621,012,754	4,968,057,848
20	4,878,666,808	39,029,188,884
21	39,333,324,973	314,666,222,712
22	336,376,244,042	2,691,008,701,644
23	3,029,242,658,210	24,233,937,684,440
24	28,439,272,956,934	227,514,171,973,736
25	275,986,683,743,434	2,207,893,435,808,350
26	2,789,712,466,510,280	22,317,699,616,364,000
27	29,363,495,934,315,600	234,907,967,154,122,000

Table 2.1: Total number of solutions for given *n* size.

Backtracking the N-Queens problem is a process of systematically placing queens and when we get to a position where no more queens can be placed, we backtrack to the previous placement and try the next available placement. In 2.1 we can see a full run through of the backtracking algorithm to a single solution. The algorithm will start by placing a queen in the first available spot in the first row. It then moves onto the second row looking to place, there are two available locations, it will pick the furthest to the left first, although as it goes to place the third queen, it will find no available placements. It will then remove the last placed queen and place it on the next available location. This continues until a solution is found although at this point the algorithm can return the board or increment the total number of solutions and then continue looking for solutions. This algorithm is complete and thus will find all possible solutions. This solution essentially creates a tree with the further you descend the tree, the more queens are placed on the board.

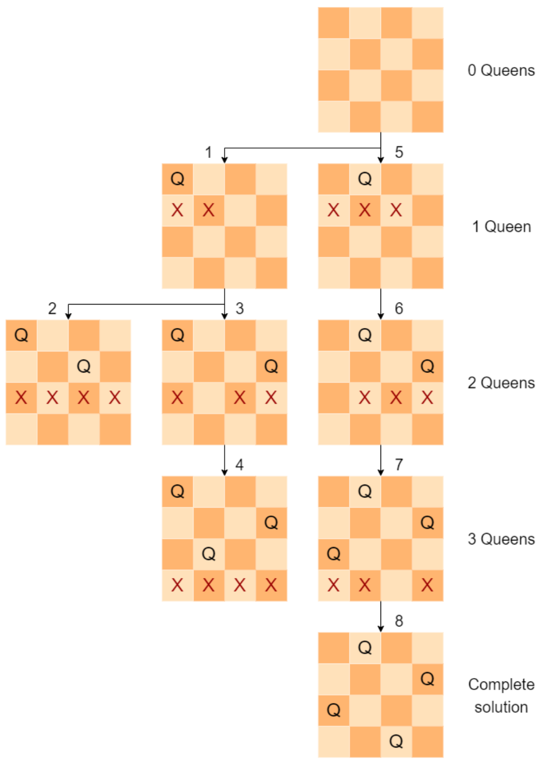


Figure 2.1: A N-Queens backtracking algorithm shown to get one solution for $n = 2$

An idea to compose a solution to the N-Queens problem by using the solutions to smaller n sizes was introduced by Ahrens [12]. Although the idea was limited by the fact it was only able to produce a strict class of solutions. Therefore it was a non complete solution unlike search based solutions.

Artificial Intelligence can also be used to solve the N-Queens problem [13]. We can begin with a randomly generated board such as one where a single queen is placed in each row/column like in the brute force example. From this point we can count the number of attacks between each of the queens and attach values to the queen placements. We can use heuristics which uses this information to improve placement of the queens and then analyses the placements again. The algorithm's speed and efficiency is heavily based on the initial configuration of queens. This is a greedy algorithm meaning it will select the best option available at the moment. This isn't ideal for generating all solutions to the N-Queens problem as greedy algorithms are not complete.

The N-Queens algorithm is an NP-complete as well as #P-complete algorithm. For a problem to be NP-complete it has to be in the set NP meaning it can be brute forced in polynomial time and can be reducible to a simulate every other NP problem. For the problem to be #P it must belong in a class of problems that can be solved in polynomial time by a deterministic Turing machine.

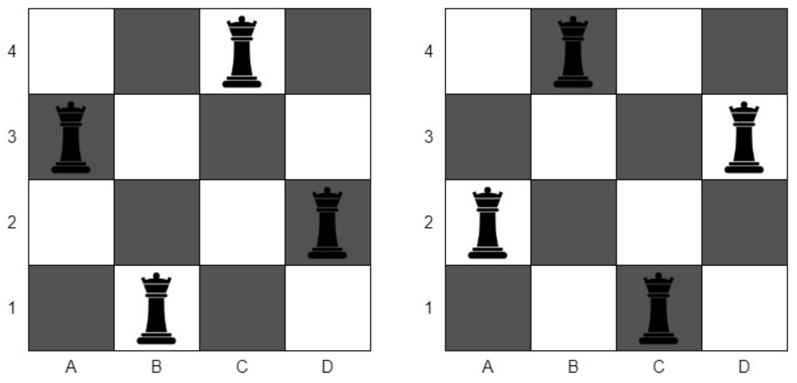


Figure 2.2: Symmetry between two 4×4 N-Queen solutions.

When looking to find all solutions for a certain size n, we can use the symmetry of the chess board to our advantage. As you can see in figure 2.2, we were able to derive a second solution to a four queen problem by reflecting the known solution. Although we have the two solutions to the 4 queen problem, only one solution is distinct and the other is a reflection/rotation. We call the distinct solution a fundamental solution. Each fundamental solution can have up to eight variant solutions (including it's original) by rotating 90, 180 and 270° and then reflecting each rotation on a fixed axis. This allows us to provide two values for the total number of solutions, the fundamental number of solutions and the total number of solutions. As you can see in table 2.1 the number of solutions increases dramatically with each increase of n. $n = 27$ is the highest order board that has been completely enumerated.

2.2 Related Work

Given the N-Queens extensive history, it has gathered a significant amount of study either looking to increase the known solutions or find real world applications. In 1986 the New Mexico State University Department of Computer Science released a paper detailing an N-Queens algorithm that could run concurrently [14]. It used the programming language Modcap which is a block-structured function-based expression language. Although they did not have access to a parallel computer, after simulating how it would run on a serial machine they came to the conclusion that it could run 300 times faster on a parallel system over serial execution.

The next documented evidence of parallelization of the N-queens problem was in 2004 by The University of Electro-Communications in Tokyo [15]. This team successfully calculated the total number of solutions to the 24-Queens problem on a 34-node PC cluster. A total of 227,514,171,973,736 solutions were calculated and took the cluster 22 days to calculate. The backtracking algorithm they used was heavily based on Jeff Somer's C program which I will go into detail about later in this paper. They designed a sequential program that is parallelized using MPI (Message Passing Interface). MPI allows for the distributed memory communication of nodes in a parallel system. I will be going into more detail about MPI later in this paper.

Later in 2004 another project surfaced that aimed to solve the 25-queens problem [16]. The research group was a group of individuals from the University of Nice Sophia Antipolis. This was achieved by using the spare CPU cycles of 260 machines using the ObjectWeb ProActive library. The total computation time was over 4,444hrs or 185 days. Although the accumulated computation time was 464,344hrs or over 53 years. The total number of solutions calculated is 2,207,893,435,808,352 and they were the first to do so. This really shows the power of parallel computing as they were able to execute the necessary tasks over 100 times quicker than on a serial system.

The $n = 26$ and $n = 27$ were both first solved by TU Dresden (Technical University of Dresdon, Germany) in 2009 and 2016 respectively [17]. To achieve this they used an FGPA (Field-Programmable Gate Array) implementation. FPGAs allows for a programmable hardware circuit that can be used to optimise a chip for a particular workload. The solution for $n = 26$ was a massive 22,317,699,616,364,044 total solutions. It took another 7 years of research from various institutions to solve $n = 27$ of the n-queens problem. It was again solved using FGPAs but these were greatly optimised and they exploited the use of symmetries to reduce the amount of computation drastically. This solution is where the record stands today with $n = 28$ which will most likely have over (10^{18}) solutions.

The N-Queens algorithm is often studied as a "mathematical recreation" although there have been several real world applications directly linked to N-Queens solutions. Erbas, Tanik and Nair introduced a memory storage concept for parallel memory system making use of the N-Queen solutions allowing conflict free access to rows, columns and diagonals [18]. Tanik also went on to describe a method for deadlock prevention through the use of N-Queens. solutions [19]. It has also been noted the N-Queens solutions could be used for VLSI (Very Large Scale Integration) testing and traffic control [13]as well as image processing [20].

2.3 Parallelisation

Parallel computing refers to the process of breaking down large computational problems into smaller parts that can be executed simultaneously across multiple processing cores that may communicate via a shared memory. We can call these smaller parts concurrent processes. Concurrent computing is the set of serial (or sequential) programs that have the ability to run in parallel [21]. The aim of parallel computing is to increase the available computational power for problem solving. The problem is first split up into discrete parts that may be solved concurrently. Each of these parts are then broken down into a series of instructions that may run serially. Each of these parts are run on separate processing units simultaneously. A control mechanism can be put in place to control what processing units execute what parts. Although parallel computing now seems like the solution to all, in many situations serial computing is quicker and more efficient than parallel computation. Parallel slowdown is a phenomenon in which more time is spent communicating with other computers than processing data causing a communication bottleneck, slowing down the system. As long as a computational problem can be split into multiple discrete parts that may run serially and can be solved without parallel slowdown, parallel computing will always be the better option.

Communication between the parallel processes is not always necessary. As stated earlier, the N-Queens problem is perfectly parallel meaning it is trivial to split into the parallel tasks. For this the only communication needed will be to add up the total number of solutions at the end of execution.

In 1992 at the Supercomputing Conference the Message Passing Interface (MPI) forum was created with the intention of creating a standardisation for message passing between parallel processes [22]. With the assistance of more than 80 people from 40 organisations, version 1 of the MPI standard was released in 1994 [23]. MPI became a communication protocol for programming in parallel with each parallel process executing in isolation. Each process will have its own dedicated memory space that the other processes do not have access to. Both process to process and collective communication is supported in the protocol. It has become the standard convention for communication among processes in distributed memory systems.

The OpenMP API used in Fortran, C and C++ supports shared memory parallel programming. It can be used to develop parallel applications that make use of the same memory space. It can be seen as multiple processes running together sharing each other's memory and resources [24]. Shared memory has its benefits over distributed memory systems with shared memory being a lot easier to parallelize compared to distributed memory.

I will be using MPI instead of OpenMP in my project for the reason that limited parallel communication is needed. If I used OpenMP, I would have to be careful with my memory management and process timing otherwise I could run into deadlock, race conditions and resource starvation [21]. I will be using the MPICH implementation of the MPI standardisation [25]. This allows me to create MPI executables in Fortran77, Fortran90, C and C++. I can then execute the program and specify the number of processors the program will use.

With that choice, I needed to decide on what language I would program this algorithm in. After doing some research I found that Fortran was much slower in parallel environments compared to C and C++ and gave a better speedup with the increase of cores [26] [27]. C++ is a super set of C meaning it is based upon C and C++ has additional functionality over C. This includes modern OOP (Object Oriented Programming) concepts such as encapsulation and inheritance. When Bjarne Stroustrup created C++, it was evolved from C and nicknamed "C with classes" [28]. As I am familiar with OOP languages and am looking to increase the set of OOP languages I understand, I will be developing this program in C++ over C and Fortran.

This algorithm will require storage of the current board state, or more specifically the queen locations. The first obvious choice would be a two dimensional matrix of booleans. We could represent 0 as an empty board location and 1 as a queen placement. Although this isn't very efficient, as there is no reason to store empty value spaces. So instead of looking at storing the current board state, we should look to store the known queen locations. Since only one queen can lie on a single row or column, we can produce an array of length n to with each value representing a column. From this we can then store the row coordinate value for each column in the array. We can then use the index of the array as the column value and the value in that index as the row value. If the board is incomplete we can use the N value to show there is no queen in that column as the possible coordinates range from 0 to the value of N (1 - 7 for $N = 8$ and we will use 8 for an empty column).

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.3: A solution the 8-Queens problem represented as a 2D boolean matrix

$$\begin{pmatrix} 7 & 1 & 4 & 2 & 0 & 6 & 3 & 5 \end{pmatrix}$$

Figure 2.4: A solution the 8-Queens problem represented as an integer array

The only other value that will need to be stored is the total number of solutions. As seen before the value of the total number of solutions increases exponentially. An unsigned long is capable of holding the total solutions for $N = 18$ (666,090,624) but not $N = 19$ (4,968,057,848). Although if we used the type unsigned long long, we would be able to store all known solutions to the problem. Although a long long is a 64 bit (8 bytes) datatype, so if executed on a 32 bit machine each integer operation would require 2 instructions inside the processor, slowing down the algorithm. If this was run on a 64 bit machine, long long integer operations could run in a single instruction and not lose any time. We could also use 2 long variables to store the solution to $N = 19$ but not $N = 20$ but again would require 2 instructions on a 32 bit and 64 bit machine.

2.4 Bit Patterns

Next we can take a look at bit pattern techniques. The main advantage of using bit operators (bitwise AND, OR, NOT and bit shifts) is the speed and efficiency of doing these operations. We still need to look at storing the queen locations. Instead of storing the exact locations of the queens, we can just store the free diagonals and columns. We will use 1 for an occupied spot and 0 for an unoccupied spot across the columns and the left and right diagonals according to the row.

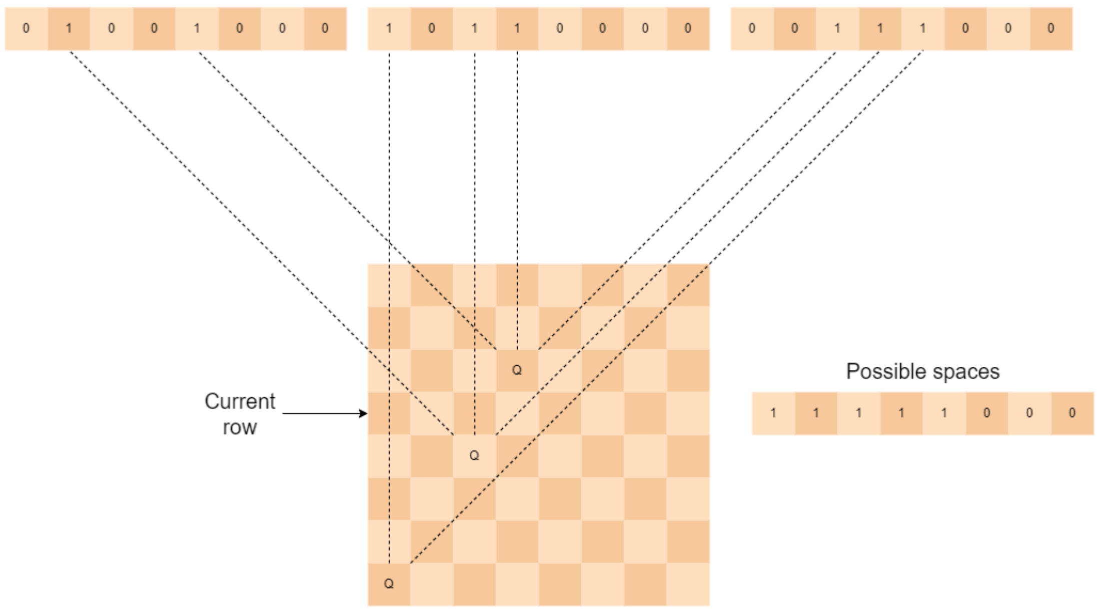


Figure 2.5: Visualisation of the available queen placements using bit patterns.

If we do a 1's complement binary operation to flip the bits, we get all the available locations for a queen to be placed in that row. This makes 1 a valid queen placement and 0 invalid. We will need a binary value the same length as the value of n. This is where we can apply a mask to the values so the length of the variables does not matter. With our previous method of 1's complement we would have a lot of 1's at the beginning of the variable outside the range of the board. This is where we can use a mask to remove these bits. The mask is initiated once we know the size of the board, and we can do a simple bitwise AND between the mask and available queen placements, to exclude the queen placements outside the boundaries of the board. This leaves us with all the available queen locations to place on the current row.

$$\begin{aligned}
\mathrm{L} &= \begin{pmatrix}0 & 1 & 0 & 0 & 1 & 0 & 0 & 0\end{pmatrix}\\
\mathrm{C} &= \begin{pmatrix}1 & 0 & 1 & 1 & 0 & 0 & 0 & 0\end{pmatrix}\\
\mathrm{R} &= \begin{pmatrix}0 & 0 & 1 & 1 & 1 & 0 & 0 & 0\end{pmatrix}\\
\mathrm{L} \mid \mathrm{C} \mid \mathrm{R} &= \begin{pmatrix}1 & 1 & 1 & 1 & 1 & 0 & 0 & 0\end{pmatrix}\\
\sim(\mathrm{L} \mid \mathrm{C} \mid \mathrm{R}) &= \begin{pmatrix}0 & 0 & 0 & 0 & 0 & 1 & 1 & 1\end{pmatrix}
\end{aligned}$$

Figure 2.6: Converting the column, left diagonal and right diagonal queen locations to available queen placements

We then want to loop through until there are no queen placements, to achieve this we loop through the available queen locations until the value is 0. When we get to 0, we will backtrack to the previous position and begin looking at the previous row. If we are on the first row, the program has completed and tested every possible queen placement.

At the start of the loop we want to reduce the possible queen placements to a single queen placement to test. In order to achieve this we can find the least significant bit (LSB) of the queen placements which is the bit that is the furthest to the right. We can easily achieve this through the use of $-x \,\&\, x$. We also want to make sure we don't attempt to place the Queen in that same spot again, so we can remove that bit from the queen placement locations by doing a bitwise XOR between the least significant bit and the queen placement locations which will remove the least significant bit.

$$\begin{aligned} x &= \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \\ -x &= \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \\ LSB = -x \,\&\, x &= \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

Figure 2.7: Calculating the least significant bit of x

After this queen placement we need to update the variables and be ready to recall the recursive function. We need to prepare the parameters before we call the function, this involves updating the queen placement location variables (left diagonals, columns, right diagonals). It's simple for the column variable as we can do a bitwise OR between the current column variable and the least significant bit. This essentially just adds the least significant bit to the current column variable, updating it for the next row. We can do the same for the diagonals but we then have to shift the bits in the same direction of the diagonal. Finally we increase the row number value, this essentially tells us how many queens have been placed. If this value ever gets to the same number as the size of the board, we have a solution and should backtrack one queen.

$$\begin{aligned} L &= \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \\ LSB &= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \\ L \mid LSB &= \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \\ (L \mid LSB) << 1 &= \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

Figure 2.8: Calculating the next row diagonal queen placements from the previous row

Jeff Somers takes the bitwise a step further by looking into the symmetry of solutions and using stacks instead of recursion [29]. The serial program is a heavily optimised backtracking algorithm as explained before and written in C. The program works by calculating one half of the solutions and then flipping the results over to the Y axis of the board. Every solution can be reflected that way to generate another unique solution. To achieve this we place 1's in one side of the column bitfield so we don't place any queens there for when we. Odd numbered boards add a region of complexity as they don't allow for reflection as you can't have 2 solutions in the same middle row/column. To to solve this, we can place a queen in the middle of the first row and then the all the next row bitfields will have 1's on one side of the odd numbered board rounded down, so if we had a 7 x 7 board the first row would look like 0001000 and the second row 0000111. At the end of execution we can double the solutions to derive the correct number of solutions. Somers also uses stacks to manage the backtracking instead of recursion. This works by using a sentinel bit that signifies the end of the stack to tell the program when backtracking is complete. The program still uses the same method to store the state of the board but uses memory allocators instead of a vector.

Board size	Number of solutions	Time to calculate
1	1	N/A
2	0	< 0 seconds
3	0	< 0 seconds
4	2	< 0 seconds
5	10	< 0 seconds
6	4	< 0 seconds
7	40	< 0 seconds
8	92	< 0 seconds
9	352	< 0 seconds
10	724	< 0 seconds
11	2,680	< 0 seconds
12	14,200	< 0 seconds
13	73,712	< 0 seconds
14	365,596	00:00:01
15	2,279,184	00:00:04
16	14,772,512	00:00:23
17	95,815,104	00:02:38
18	666,090,624	00:19:26
19	4,968,057,848	02:31:24
20	39,029,188,884	20:35:06
21	314,666,222,712	174:53:45

Table 2.2: Time to calculate the total number of solutions using Jeff Somers' program n size.

Somers did some timed tests of the program on his 800MHz pc ans successfully calculated the time to solve up to $n = 21$ [29]. The algorithm is approximately $O(n!)$ and times to calculate further solutions increases significantly. Somers predicted the solution for the $n = 22$ will take 8.5 times longer than the solution for $n = 21$ or roughly 8 and half weeks. Even a computer with increased power would struggle to get the time under a month. Although this is a completely serial program and parallel implementations will have vast time improvements. At the time of release this program was nearly twice as fast as the current world record N-Queens solution program. Somer's program is also up to 10 times as fast as Dr. Rofle's program which also makes use of symmetry [30]. At the time of creation, Somer's program was the fastest serial N-Queens program known and The University of Electro-Communications in Tokyo went on to parallise this program and use it to first solve the $n = 24$ Queens problem [15].

Chapter 3

Project Management

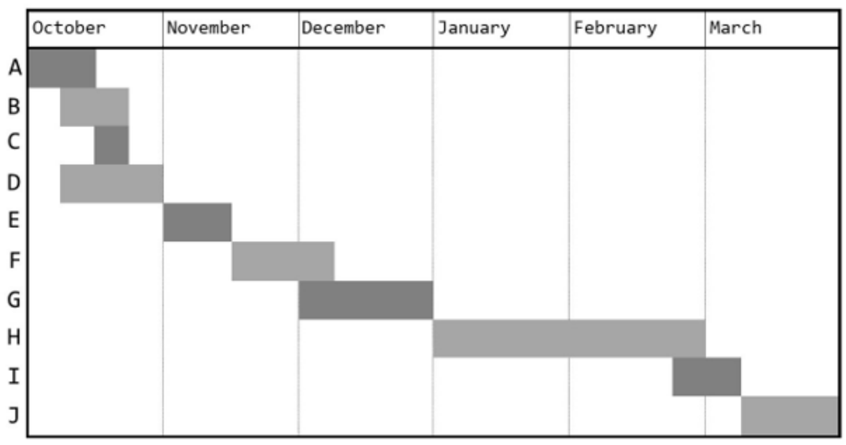
3.1 Time Management

During the writing of my initial document I came up with a Gantt chart to manage my time during the project. I was unable to maintain this due to an oversight in planning. I was not under the impression that I was able to test my parallel program on my own personal computer. Although after learning that it was not necessary to run on the HPC, I neglected Aim 1 which was to implement a parallel N-Queens algorithm on a HPC. Therefore a lot of my time was spent improving my program on my personal machine and working on meeting the other 4 aims.

It also took me a significantly longer time to learn the C++ programming language while completing the rest of my studies to my best ability. I was hoping to have a serial N-Queens prototype in C++ by the end of November although this was not met until January. I spent most of my spare time in November and December learning the basics of C++ and whether it was the correct language to use. I did some research on other common parallel programming languages such as C and Fortran but came to the conclusion that C++ was best for the job in hand which is explained in section 2.3.

With the delay of learning how to use C++, it wasn't until late February that I was beginning to use MPI. It took me a couple of weeks to research the difference between MPI and openMP and come to the conclusion of MPI which is explained in section 2.3. It again took me longer than expected to implement parallel functionality into the program, mostly because I want to learn the process, instead of looking up another person's implementation. A lot of this paper will be spent discussing the process in detail rather than the result.

To reflect on the timeline, I would say that not enough research was put into how long these tasks would take. This has led to an unachievable timeline with no impact on my other studies. If I were to plan this project out again, there would be a lot more time for the research element of this project so less trial and error in programming would be needed. I would also plan to learn C++ before starting a project of this size, mainly in that language.



A	Research of N-Queens problem
B	Research of parallelization
C	Working serial N-Queens prototype in Java
D	Initial document
E	Working serial N-Queens prototype in C++
F	Running parallel program on HPC
G	Running parallel N-Queens program on HPC
H	Improving parallel N-Queens program on HPC
I	Final dissertation outline
J	Final dissertation

Figure 3.1: Gantt chart with labels

3.2 Risk Mitigation

As stated in my initial document, I will have access to over £16m worth of computational resources. Their facility has a total of 13,080 processor cores capable of delivering one petaflop of computational power [31]. To be able to use their facilities I must abide to a set of terms and conditions as well as maintaining a good standard of conduct.

Supercomputing Wales does not take responsibility for data generated or stored on their system, so I must make sure that backups are made of all programs and solutions run on their systems. In order to achieve this I will use Git version control to backup my programs. Should I experience any issues with my own computer I will have the code backed up on a private repository on GitHub. This will nullify any issues I should face with data loss. If I were to break or lose access to my personal computer, having these backups on GitHub would allow me to continue working on my project on an alternate computer. I can transfer data from the HPC system using filezilla and I made use of this instead of using git commit from their system [32].

I paid close attention to how much processing power I really require, as requesting more processing cores and memory than required can result in penalties for my account. I must also respect other users of the hardware and not run too many jobs in a short space of time and not to fill up their storage drives.

Again any breakages of these rules may result in penalties or removal of my account.

3.3 Testing

As I am executing code on equipment that I don't control, I must confirm that my code runs correctly bug free. Although not a massive amount of testing had to be carried out, I made sure to test a variety of inputs in the program itself. On each execution the program tested whether the output value was correct and return the result to a file. As the N-Queens is a well documented computational problem, the solutions up to $n = 27$ are readily accessible. There is no reason to keep the solution, so it is discarded at the end of execution. Before running any code on the HPC I would ensure the code compiles and runs on my PC. If I know the run time won't be significant, I will run the duration of the program on my PC. By the end of the project my code would always produce the correct value to a solution. I will go into detail about automatic testing later in the paper.

Chapter 4

Implementation

This project is an exploration into parallel computing and looking into the history of a famous computational problem. I also wanted to use this project as an excuse to learn a new programming language and develop my computer science skills. To achieve this I researched multiple methods of solving the N-Queens problem and multiple parallel computing techniques and protocols.

4.1 N-Queens in Java

To begin with I attempted to solve the N-Queens problem in the programming language Java. This is because I am very familiar with the language and believed having a better understanding of the algorithm would assist me in translating N-Queens into an unknown language. I began programming this before any research into the algorithm and how to solve it. I saw the algorithm naturally tended itself towards backtracking and so I attempted to use recursion to solve the problem. My first attempt was very inefficient as I stored the board as a two dimensional array instead of a one dimensional or bitwise as explained in my project background. My function to check whether a square was free to place a queen was also very inefficient. For all the rows, columns and diagonals the program would loop though the board, instead of sharing the same board loop. Once the program worked I did begin to optimise and quickly spotted this error as well as reducing the size of the board from a 2D array to a singular one dimensional. The program was not developed any further than this, as I did not intend to use it in a parallel setting, it was merely a learning experience to help me understand the problem better. I was only able to realistically calculate up to the $n = 16$ solution and that would still take upwards of 30 minutes to compute.

4.2 Serial N-Queens in C++

After the success of my program in Java, the next logical move was to attempt to translate this code into C++. I have no prior experience in this language, so first and foremost I attempted to program a series of simpler programs which tackle similar problems. Examples of these programs included simple file reading and writing, programs that worked with 2D arrays and a recursive Fibonacci generator. The weeks I spent writing other programs and reading tutorials benefited me greatly when it came to starting the N-Queens program.

I first attempted to translate the inefficient version of my N-Queens program into C++ with the 2D arrays as I deemed it simpler. I struggled with the concept of pointers and getting the arrays to function correctly. After a week or so of tinkering, I was able to produce a program in C++ that could find all the solutions to the N-Queens problem. Soon after this I worked on improving the efficiency of the algorithm in the same way I did in Java. This was simple to achieve as after working with 2D arrays before, 1D arrays were much simpler to work with. If I did this again, I would start with implementing the 1D arrays as the 2D arrays caused me a lot of issues.

The next part was to add functionality to generate sections of the backtracking tree that can be run together then added up to produce the correct number of solutions. As the N-Queens problem is perfectly parallel, it is trivial to split it up into these sub-problems. To achieve this we can run our N-Queens algorithm but instead of adding a solution once the board is full, we can return the current board state once a certain number of queens has been placed. Then once we want to run the subprogram, parse the incomplete solution and the program will attempt to solve it, but won't backtrack past the input board state. This board state was simply written to file, no other data like the board size is needed as we can derive it from the board state.

I had an issue with running the algorithm with the new board states. I was using the `std::array` container type which has a fixed length. All possible arrays I used had a fixed length, as the board size would not change and neither would the total number of generated permutations. Although this had to be a value that is specified during the compiling of the program. This is not ideal for my code, as I intend to run multiple values of *n* and do not want to hard code in values and have to recompile before each execution. To solve this I changed my container type to `std::vector` which is essentially a dynamic array. You may add items to the vector by using the push back function which will automatically add a value to the vector and handle any storage changes.

I can now simulate running this program in parallel by running each section of the backtracking tree serially and when the results are added up they still provide the correct answer. In order to check that I have the right answer, I created a `std::map` container that holds the correct value for each value of N. Now instead of checking if the number of solutions is correct manually, the program will automatically check if their calculated solution is correct and return true or false respectively.

4.3 Parallel N-Queens on my PC

The next step is to implement the MPI standardisation and begin to run the sub programs in parallel. Again as this problem is perfectly parallel, little communication is needed between the processes. Each MPI program in C++ has a set structure that it must follow in order to work. Firstly the MPI header file must be included with `\#include <mpi.h>` which contains the MPI function type declarations. According to the MPI documentation the main function at the start must begin with `MPI_Init(\&argc, argv);` which initialises the MPI environment. At the end of the main function and at the end of the parallel execution comes `MPI_Finalize();`. The code in between these 2 commands will run on each parallel process and if you were to print a value, it would print the number of times as the number of processors being used. So far we cannot determine what processor we are currently running on. We can use the command `MPI_Comm_size(MPI_COMM_WORLD, &worldsize);` to get the number of processors the program is running to the variable worldSize. The command `MPI_Comm_rank(MPI_COMM_WORLD, &myRank);` will get the rank of the current processor to the variable myRank. We now have the issue where if we want to print a value, it will always print the same number of times as the total number of processors. To avoid this we can use an if statement to check that `&myRank` is 0 (The first rank is always 0) and then we can run code only on the first rank.

From this we can run a very basic example by manually telling each rank what to run. We would then have to manually add up all the results at the end which isn't ideal. The command `MPI_Reduce()` reduces the values on all ranks to a single value. We can use this to add all the total values together at the end of execution. Although we must wait for all the processes to finish their execution otherwise we might add up the total solutions before a process has finished calculated resulting in an incorrect total number of solutions. To solve this we can use the command `MPI_Barrier(MPI_COMM_WORLD);` which will block all processes from continuing until every process has reached this point in the code. If we put the barrier before the reduce command we can be certain that all processes have finished their execution before adding up the solutions.

Next we want to automatically assign tasks to the processes so we don't have to edit the code for each increase of processors. To achieve this we can loop through the processes as a vector that increments by the total number of processors. If we then add on the current rank value we will get an index for a list that is unique for each processor. If we had 4 tasks on 2 processors, the first loop would yield us 0. The first processor would add on its rank which is 0 yielding 0 and therefore it would run the task at index 0 on the list. The second processor would add its rank onto 0 giving 1 and then run the task at index 1 on the list. In the second iteration of the loop it would increment by 2 as there are 2 total processors and the index value would be 2. Processor 1 would then run the task at index 2 and processor 2 would run the task at 3 and fully complete the list of tasks. This can be scaled up to any number of processors and tasks although we must check if the index lies within the list as if the length of the task list is not divisible by the total number of processors, some processors will do 1 less task than the rest.

4.4 Parallel N-Queens on a HPC

The next logical step is to get the program running on the HPC. Before this was done, extensive bug testing was completed to ensure that the algorithm had a 100% success rate. I had created a shell script to automatically test various values of N, depth and processor counts. I was only able to run up to 4 processors on my personal PC as that was all that was available. I used the MPICH implementation to compile and run my code. I used the command `mpiCC inputFile.cpp -o outputFile` to compile the code. When running the code I use the command mpirun. To specify the number of processors I used the tag `-n` followed by the number of processors to be used. I will go into more detail about my shell script for testing later in the paper.

In order to communicate with the HPC, Supercomputing Wales recommends you use PuTTY which is a SSH and Telnet client [33]. It allows me to securely connect to their HPC and run commands from my personal computer. Supercomputing Wales uses the SLURM workload manager to assist in scheduling jobs and managing clusters for their system. In order to run programs on their system I require a job script that details the resources I need and the files I want to run as a job. Before running the job file I will need to recompile my programs within their system. I have a vast choice of compilers and MPI implementations and if I had more time in this project, I would look to test my program in these different compilers and measure the difference in speed between each one. I can install these compilers and additional modules as I need them into the terminal or in the job script. As stated earlier I used the MPICH implementation on my PC to test and will be using it again to execute code on the HPC. Specifically I will be using the `mpi/mpich/3.2.1` library to execute my code by running the `mpiexec` command. I will be using the Intel implementation more specifically the `mpi/intel/2020/0` version from their available libraries o compile my code. I had issues with compiling the code at first as I am using features from C++11 and the default compiler was not using this version. To circumvent this when compiling I used the tag `-std=c++11` which allowed me to compile using features from that version. The full command to compile the program for the HPC was `mpiicpc -std=c++11 inputFile.cpp -o outputFile`.

Chapter 5

Results

When running the N-Queens program I am only concerned about whether the program operated correctly and how long it took to execute. I explained how I used the map container type to verify that my program was producing the correct number of solutions and therefore I just added a 0 to the output file if the test was successful and 1 if it was not. I also would print all the relevant information about the execution parameters such as the size of board, depth of initial tree and how many processors the program ran on.

There are a variety of techniques to return the time taken to compute. In my first Java and C++ program I calculated the time to run inside of the program. I did this in C++ by using the time standard library. You are able to start the clock whenever and when stopping it can be formatted into whatever form necessary. Although I can only get the total time it took from the start to finish and not any extra details.

I was introduced to the Linux command `time` that when placed before a command will print the real, user and system time. Real time is the real time that the program or command took to run this includes time in which processes may be blocked. This is the same as the previous technique but begins calculating the time as soon as the command to run the program has been executed and until it has completely finished execution. User time is the total amount of time that the CPU spent in user mode which is outside of the kernel. This is only the actual CPU time spent executing the process. Finally system time is the amount of time spent in the kernel within the process. The kernel will be responsible for commands that include memory allocation and hardware access. Both system and user time are not easily obtainable from the program itself making use of the time command better.

Up until this point I was developing this program on my laptop running Windows 10. I spent a while looking for an alternative time command to use in Windows, but was not able to find one with the same functionality. I was lucky enough to have a spare desktop computer lying around and installed Kubuntu and continued development from this point in Linux. As all my files have been backed up to Github moving the development to a new computer was no issue.

I was now able to run the time command at the start of my program and wrote the result of it to a file.

Although the default format included what time it was measuring and not just the value (`Sys: 0.01` instead of `0.01`). This would make it difficult to work with this data afterwards and abstraction would be required. By default there was no way to abstract what type of time it was although after researching I discovered a slightly different command. By using the GNU version of the command which can be run with `/usr/bin/time`. An alias can also be used to overwrite the default time command (`alias time=/usr/bin/time`). The main difference between using the default and GNU, is the fact you can format the output string with a series of flags. From this I was able to format the string using `-f "%E %U %S"` which would return the real, user, and system time with a space dividing them. I was able to then use the flag `-a -o testResults.txt` which would append the results to the end of the output file. I did not need to create a new line in the file because the C++ program creates a new line when a test is complete.

From there I can create the automatic tester shell file that will loop through all the parameters I would like to test and generate the depth values before running the algorithm. Ideally I would run each test as many times as possible and then take the lowest time result. I can take the lowest possible time because it is not a random program and any differences in the times is due to minor differences in the hardware availability when the code ran.

As some tests were run multiple times, I made a simple python program that will only take the lowest time result for each of the duplicate tests. At this point I would remove any failed tests, but I had a 100% test success rate. This was formatted back into a file with all the relevant information needed and all the times converted to seconds for ease in plotting.

5.1 PC Backtracking Results

I ran tests on both my personal computer as well as the HPC. This is because I can leave my computer running for days at a time and not consume shared resources. Although I was only able to use up to 4 processors and for the array backtracking solution I only ran tests up to $n = 14$ as tests from that point took upwards of a minute and I was planning on running a few hundred tests per board size.

In figure 5.2 I plotted the real time of the program against the value of N being parsed into the N-Queens problem algorithm. I plotted a line for each of my processor tests (1, 2 and 4 core) although I believe the results for 4 cores are incorrect. This is because we should see an improvement over the use of 2 cores but there is none. I believe the reason for this is the MPI implementation can only run 2 cores on my particular PC. This is not a big issue as the results from the HPC will not have this issue.

In figure 5.2 I plotted the real time of the program against the depth of tree initialisation. The time here does not include the time needed to generate the tree. We can see a clear picture that generating the tree beforehand does improve the run time of the algorithm and it begins to have large effects on the 14-Queens problem after a depth of 9.

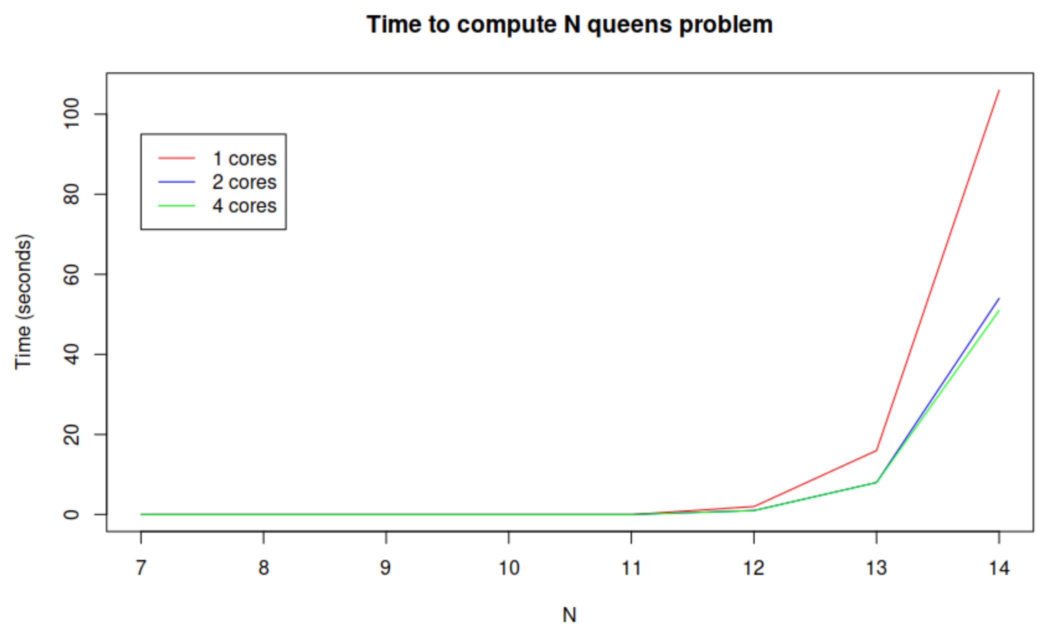


Figure 5.1: Run time of the backtracking N-Queens program plotted against the value of N run on PC

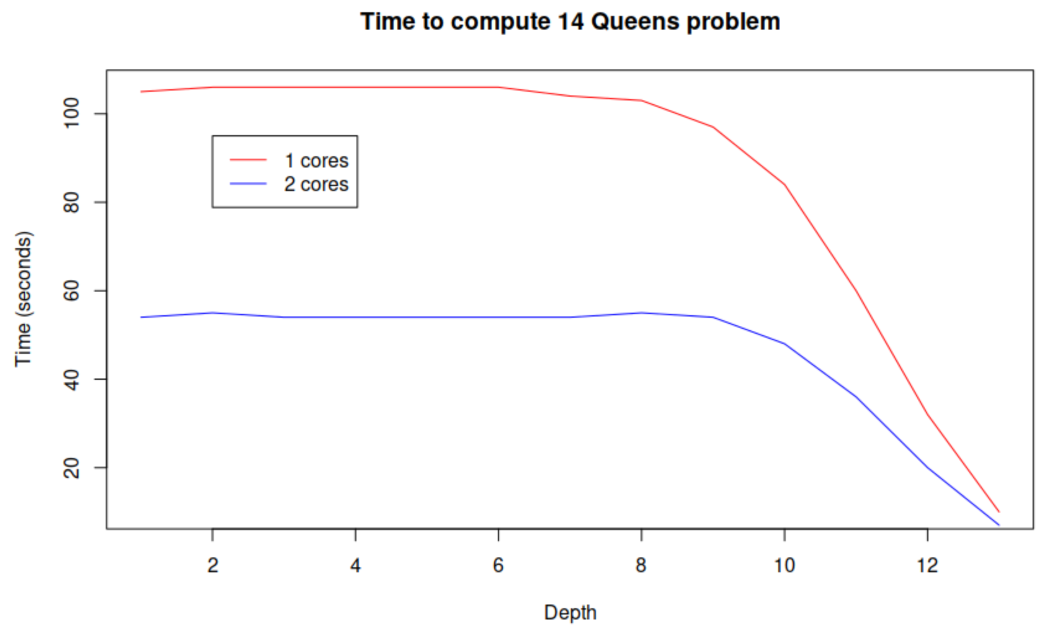


Figure 5.2: Run time of the program plotted against depth of initial tree to solve the 14-Queens problem

5.2 HPC Results

As you can see in figure 5.3 I plotted the real time of the program against the value of N being parsed into the N-Queens problem algorithm. This is the same graph I plotted on my PC which had errors with running 4 processors. I theorised this was because of my MPI implementation and not the result of the program. We can see in the graph that I was correct and the program worked as intended and 4 processors had a speedup over 2 processors.

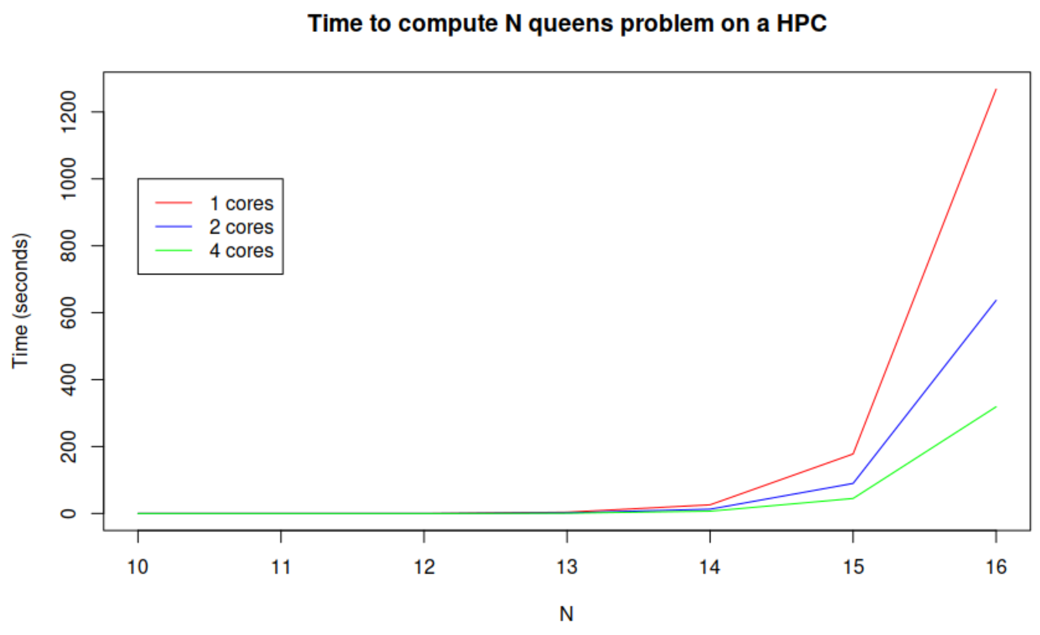


Figure 5.3: Run time of the backtracking N-Queens program plotted against the value of N run on HPC

In figure 5.4 I plotted the time to compute the solution to the 16-Queens problem against the processors used. I got data from processor counts 1, 2, 4 and then every multiple of 8 until 64. We can see a significant speedup with the early increases in processor count, but less so with the larger increases and this is to be expected because of Amdahl's Law. Amdahl's law states that the overall performance improvement gained by optimising an individual part of a system is limited by the fraction of time that the improved part is actually used.

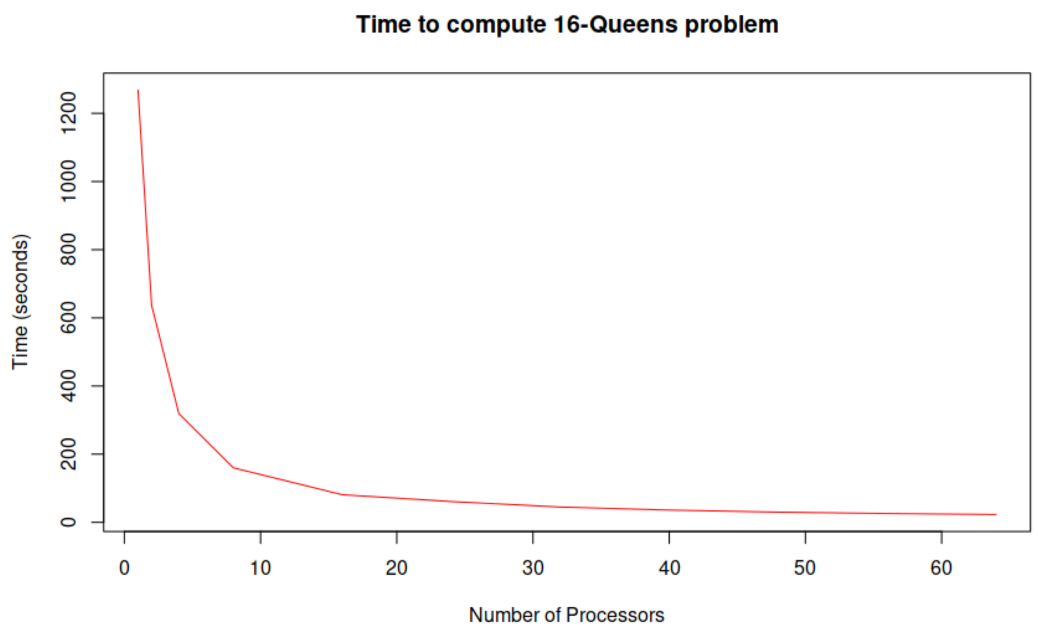


Figure 5.4: Run time of the backtracking 16-Queens program plotted against the number of processors used

5.3 Bitwise Results

Board Size	Number of Solutions	Bit Patterns (hh:mm:ss)	Arrays (hh:mm:ss)
10	724	< 0 seconds	< 0 seconds
11	2,680	< 0 seconds	< 0 seconds
12	14,200	< 0 seconds	< 0 seconds
13	73,712	< 0 seconds	< 0 seconds
14	365,596	< 0 seconds	00:00:02
15	2,279,184	< 0 seconds	00:00:09
16	14,772,512	00:00:01	00:00:45
17	95,815,104	00:00:03	00:06:16
18	666,090,624	00:00:21	00:47:59
19	4,968,057,848	00:02:42	N/A
20	39,029,188,884	00:21:28	N/A

Table 5.1: Time to calculate the total number of solutions using arrays and bit patterns.

In table 5.1 we have the real time for 2 different N-Queens algorithms. The first makes use of Arrays and the second makes use of bit patterns. Both ran on the HPC at Supercomputing Wales, used 32 cores and all board sizes had an initial configuration depth of 6. As you can see from the data the bit pattern algorithm was up to 100 times faster in some cases than the array method.

Board Size	Number of Solutions	Bit Patterns (mm:ss:ms)	Somers (mm:ss:ms)
10	724	< 0 seconds	< 0 seconds
11	2,680	< 0 seconds	< 0 seconds
12	14,200	< 0 seconds	< 0 seconds
13	73,712	< 0 seconds	< 0 seconds
14	365,596	< 0 seconds	00:00:01
15	2,279,184	00:00:02	00:00:04
16	14,772,512	00:00:11	00:00:23
17	95,815,104	00:01:20	00:02:38
18	666,090,624	00:09:55	00:19:26

Table 5.2: Time to calculate the total number of solutions using bit patterns and Jeff Somers' implementation.

I've compared my results to Jeff Somers' program in table 5.2. I ran my bit pattern program on the HPC but with a single processor as Somers' program is not parallel. The results show that my program was quicker than Somers, but there is a stark difference in hardware. Somers ran his code on his 800MHz PC while my code was run on the HPC which is running at 2.4GHz so we would expect a 3 times speedup, whereas in reality there is only a 2.

Running previous graphs from before on the bit pattern algorithm show similar graph patterns and little is learnt from them with the existence of the previous graphs.

Board size	Number of solutions	Time to calculate (hh:mm:ss)
10	724	< 0 seconds
11	2,680	< 0 seconds
12	14,200	< 0 seconds
13	73,712	< 0 seconds
14	365,596	< 0 seconds
15	2,279,184	< 0 seconds
16	14,772,512	00:00:01
17	95,815,104	00:00:03
18	666,090,624	00:00:12
19	4,968,057,848	00:01:18
20	39,029,188,884	00:10:14
21	314,666,222,712	01:26:15

Table 5.3: Fastest time to calculate the total number of solutions using bit patterns

In table 5.3 I gathered the quickest result for each size of board. The highest solution value I completely enumerated was $n = 21$ which ran in an hour 26 minutes and 15 seconds. Acquiring this solution on the alternate algorithm would be nearly impossible with my time and resources.

Chapter 6

Conclusions and Future Work

I have been able to achieve my first aim "To implement a parallel N-Queens algorithm on a HPC.". I successfully implemented 2 parallel N-Queens algorithms, the first of which was a backtracking solution that made use of a size N array to hold the board and the second of which was another backtracking solution which made use of bit patterns.

The second aim was to "To measure and collect the speed and accuracy of the algorithm." which I was able to achieve. Each test run of the program would automatically test if the result is correct and label the result accordingly. I was able to measure multiple types of speed data including real, user and system time. Again this data was stored in a text file in a format I defined alongside all the parameter data.

The next aim was to "Use the speed and accuracy data collected to improve and redesign the algorithm.". During the project I improved small aspects of each program after each execution. I tested various techniques of doing things such as checking the board for queen locations, and then picking the most accurate and quickest option to go in my program.

The fourth aim is to "Summarise my findings and compare the various algorithms and methods implemented.". I conducted a few thousand tests with many different parameter changes such as the program parameters, hardware parameters and trying different algorithms. The program parameters were the board size, processor count and the depth of the initial tree. The hardware parameters included running on my personal computer and the HPC. I summarised these tests as graphs and smaller tables by picking key variables to compare. I was able to test 2 different algorithms, if I had extra time I would investigate another method of solving further such as artificial intelligence.

Finally I wanted "To correctly calculate the highest value of n and to compare my results to other projects.". The highest value of n I was able to compute was $n = 21$ in which the solution was 314,666,222,712. I achieved this by running the parallel bitwise algorithm on the HPC and it took 1 hour 26 minutes and 15 seconds running on 64 cores. I'm sure I could use the same program to calculate the solution to the 22-Queens problem in under 24 hours and even the 23-Queens problem in under a week. I wanted to maintain the code of conduct for Supercomputing Wales and be aware of other users on the platform and so did not run the 22-Queens and 23-Queens.

The results show the importance of parallel computing and how it can be used to get solutions to problems that may have been impossible without it. This project has taught me significant amounts about parallel computing and optimisation. Before I began this project I knew very little about the parallelisation of perfectly parallel serial programs and since then I believe I have a firm grasp on the subject.

6.1 Future Work

If I were to continue work on this project I would like to compare these 2 algorithms to another. An idea would be to research and implement an algorithm similar to Jeff Somers. I didn't make use of symmetry during this project and it's a trait of the problem that can be exploited for time reduction. I would also want to test the programs I have with different compilers and libraries. Finally I would like to experiment with the parsing of data into the parallel tasks. I currently put all the values into an array and parse them in 1 by 1, but I'm sure there are other more efficient solutions that don't require resource heavy arrays. There are so many parameters to the N-Queens problem and such a variety of solution techniques that no 1 paper can give it justice.

If I were to restart the project knowing what I know now, I would put a lot more time into learning C++ and how it can be optimised. I feel I lost a lot of possible optimisation because I didn't know that it existed. I would have also liked to have spent more time researching the current history of the problem and seeing what people are pursuing now to solve the 28-Queens problem. With the popularity of quantum computing increasing, it would be interesting to research whether there is any possibility of using it to solve the N-Queens more efficiently.

Overall I am pleased with how the project panned out and I am pleased with the result. This paper has given an insight into the development of a parallel program from square one. I was able to see a speedup with each core increase on both my algorithms when run on the HPC.

Bibliography

[1] A. Krizhevsky, “Proposal of 8-queens problem,” *Berliner Schachzeitung*, vol. 3, no. 363, p. 1848, 1848.

[2] F. Nauck, “Briefwechseln mit allen für alle,” *Illustrirte Zeitung*, vol. 15, no. 377, p. 182, 1850.

[3] E. J. Hoffman, J. C. Loessi, and R. C. Moore, “Constructions for the solution of the m queens problem,” *Mathematics Magazine*, vol. 42, no. 2, pp. 66–72, 1969. [Online]. Available: https://doi.org/10.1080/0025570X.1969.11975924

[4] E. W. Dijkstra, *A short introduction to the art of programming*. Technische Hogeschool Eindhoven Eindhoven, 1971, vol. 4.

[5] K.-B. Li, “Clustalw-mpi: Clustalw analysis using distributed and parallel computing,” *Bioinformatics*, vol. 19, no. 12, pp. 1585–1586, 2003.

[6] “Top500.” [Online]. Available: https://www.top500.org/

[7] “Top500 supercomputer fugaku,” 2021. [Online]. Available: https://www.top500.org/system/179807/

[8] S. W. Golomb and L. D. Baumert, “Backtrack programming,” *Journal of the ACM (JACM)*, vol. 12, no. 4, pp. 516–524, 1965.

[9] B. A. Nadel, “Constraint satisfaction algorithms 1,” *Computational Intelligence*, vol. 5, no. 3, pp. 188–224, 1989.

[10] X. Hu, R. C. Eberhart, and Y. Shi, “Swarm intelligence for permutation optimization: a case study of n-queens problem,” in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS’03 (Cat. No. 03EX706)*. IEEE, 2003, pp. 243–246.

[11] B. A. Nadel, “Representation selection for constraint satisfaction: A case study using n-queens,” *IEEE Intelligent Systems*, vol. 5, no. 03, pp. 16–23, 1990.

[12] W. Ahrens, *Mathematische unterhaltungen und spiele*. BG Teubner, 1918, vol. 2.

[13] R. Sosic and J. Gu, "A polynomial time algorithm for the n-queens problem," *ACM SIGART Bulletin*, vol. 1, no. 3, pp. 7–11, 1990.

[14] R. Silver, M. Wells, S.-I. Wu, and M. Hug, "A concurrent n 8 queens' algorithm using modcap," *SIGPLAN Not.*, vol. 21, no. 9, p. 63–76, Sep. 1986. [Online]. Available: https://doi.org/10.1145/885694.885703

[15] K. Kise, T. Katagiri, H. Honda, and T. Y, "Solving the 24-queens problem using mpi on a pc cluster," 2004. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.4098&rep=rep1&type=pdf

[16] D. Caromel, C. Delbé, A. Di Costanzo, and M. Leyton, "Proactive: an integrated platform for programming and running applications on grids and p2p systems," *Computational Methods in Science and Technology*, vol. 12, pp. issue–1, 2006.

[17] T. Preußer, B. Nägel, and R. Spallek, "Putting queens in carry chains," 10 2021.

[18] C. Erbas, M. M. Tanik, and V. Nair, "A circulant matrix based approach to storage schemes for parallel memory systems," in *Proceedings of 1993 5th IEEE Symposium on Parallel and Distributed Processing*. IEEE, 1993, pp. 92–99.

[19] M. M. Tanik, *A graph model for deadlock prevention.* Texas A&M University, 1978.

[20] C.-N. Wang, S.-W. Yang, C.-M. Liu, and T. Chiang, "Fast motion estimation using n-queen pixel decimation," Apr. 11 2006, uS Patent 7,027,511.

[21] M. Ben-Ari and M. Ben-Arî, *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[23] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: the MPI core*. MIT press, 1998, vol. 1.

[24] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[25] W. Gropp and E. Lusk, "User's guide for mpich, a portable implementation of mpi," 1996.

[26] L. E. Young-S, D. Vudragović, P. Muruganandam, S. K. Adhikari, and A. Balaž, "Openmp fortran and c programs for solving the time-dependent gross–pitaevskii equation in an anisotropic trap," *Computer Physics Communications*, vol. 204, pp. 209–213, 2016.

[27] J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. Reynders, and P. J. Hinker, "Comparison of c++ and fortran 90 for object-oriented scientific programming," *Computer Physics Communications*, vol. 105, no. 1, pp. 20–36, 1997.

[28] B. Stroustrup, "A history of c++ 1979–1991," in *History of programming languages—II*, 1996, pp. 699–769.

[29] J. Somers, "The n queens problem a study in optimisation." [Online]. Available: http://users.rcn.com/liusomers/nqueen_demo/nqueens.html

[30] T. Rolfe, "Queens on a chessboard: Making the best of a bad situation," in *Technical Paper Sessions at the Small College Computing Symposium at Augustana College (Sioux Falls, SD)*, 1995, pp. 21–22.

[31] "Supercomputing wales." [Online]. Available: https://www.supercomputing.wales/

[32] "Filezilla." [Online]. Available: https://filezilla-project.org/

[33] S. Tatham, "Putty: a free ssh and telnet client." [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/putty/