Westfälische Wilhelms-Universität Münster

Bachelor thesis

Collaborative Randomized Incremental Constructions on the Examples of N-Queens

Author:
Tristan Menke
t_menk05@uni-muenster.de
Matr. Nr. 429262

Supervisor:
Florian Fey
feyf@uni-muenster.de

submitted on 2nd May 2020

Fachbereich 10
Mathematik und
Informatik

Abstract

A lot of computational problems can be stated as Constraint Satisfaction Problems, including many games, optimization problems and various artificial intelligence tasks. This thesis presents a way to find valid solutions for them through randomization, called Randomized Incremental Constructions. Finding a single valid solution can be sped up through parallel computation, by executing many instances with different random choices concurrently. Usually, Randomized Incremental Constructions reset any progress made in case of failure. This thesis studies a way to improve parallel computation of Randomized Incremental Constructions, by sharing promising solutions between threads in a collaborative manner instead of resetting. Constraint Satisfaction Problems are similar in nature, and finding a fast, general way to solve them has uses in many areas. To test and compare any approaches presented, we use the frequently studied N-Queens problem as a specific Constraint Satisfaction Problem. Based on several randomized tests, we have found that collaborative Randomized Incremental Constructions are preferable to non-collaborative execution on modern CPU and GPU architectures.

Table of Contents

1 Introduction

The following is the bachelor thesis for computer science, written by Tristan Menke in the winter semester of 2019/2020 at the University of Münster. This thesis was supervised by Florian Fey. This document has been written built on a modified version of a template by Jannes Bantje [Ban16].

To begin, this thesis gives an overview over Constraint Satisfaction Problems, an interesting class of problems that have many real-world applications, and how they have been solved historically by an algorithm called Backtrack Search. To evaluate the approaches presented in this thesis, we introduce the N-Queens Problem as a Constraint Satisfaction Problem. Following that, there is an overview of a simple randomized approach, called Randomized Incremental Constructions as a superior alternative. To motivate further research, an overview over some problems of high-performance computing is given, which necessitates the use of parallel algorithms. Motivated by that, we present how Randomized Incremental Constructions can be run by many threads concurrently to speed up the convergence to a single valid solution. The heart piece of this thesis is an improved version of parallel Randomized Incremental Constructions, that we call Collaborative Randomized Incremental Constructions. We propose it as a superior form of Randomized Incremental Constructions, that is just as generally useful as the simpler algorithm. For this, we present some possible implementations and then compare them to the naive Randomized Incremental Constructions approach. First, implementations for CPU architectures for both are presented and compared through a series of tests. Following that, we implement and test both approaches on a GPU to see if they might be used on that architecture. GPU tests and their results, as well as conclusions about the use of Collaborative Parallelism conclude this thesis.

2 Constraint Satisfaction Problems

Many computational tasks can be interpreted as a Constraint Satisfaction Problem (CSP). A CSP is defined by three components as explained in [DP87]:

- A set of N variables $X_1, ..., X_N$
- A set of domains $D_1, ..., D_N$, one for each variable, where D_i defines the values the variable X_i may assume.
- A set of constraints that restrict the values variables in the set may take. Formally, a constraint is a relation between a local collection of variables. They may constrain one or more variables. A binary constraint constrains two variables and would be: $R_{i,j} \subseteq D_i \times D_j$. Constraints restrict the values that variables may take at the same time, making them a subset of the Cartesian Product of the variable domains. For example, a constraint $R(X_1, X_2, X_3)$ could specify that $X_1 * X_2 = X_3$ must be true. Constraints are often presented in the form of equations or inequalities, but do not have take such a form.

A solution to a CSP is an assignment of all the variables $(X_1, ..., X_N)$, such that all constraints for all the variables are satisfied. Finding solutions to CSPs in general is an NP-Complete problem, but specific CSPs may be easier to solve [AM15]. CSPs are interesting because a wide variety of computational tasks can be formulated as a CSP. Finding algorithms to solve CSPs in general, or solving a specific CSP in a way that can easily be adapted for another CSP, can thus help solve a lot of real problems. Furthermore, if it is possible to solve all CSPs generally, quickly and in an easy to implement way, it is possible to solve new CSPs without having to look into specific solutions. Some CSPs have no requirements to be solved quickly, or are not large enough to warrant a fast algorithm, such as simple games like Sudoku [Kem18]. But some applications also require efficient strategies for finding solutions either because they are time critical, as in the case of scheduling for operating systems or traffic control, or because they require processing of many constraints as in linear optimization problems, classification and artificial intelligence. Many applications of the N-Queens Problem, which is a CSP, are presented in [BS09]. Because of their varied real-world use, it is important to find efficient general algorithms that work well in practice. To do this, we must also use modern hardware to its fullest potential.

2.1 The N-Queens Problem

The N-Queens problem is a general version of the 8-Queens problem. These problems have been researched since before computers have been invented, originally being of interest to mathematicians and chess players, such as Gauss [Ahr01].
In this problem, the goal is to arrange N queens on an NxN chess board, such that no queen can threaten another queen. This means that each row, column and diagonal must contain at most one queen.

In computer science, the N-Queens Problem is a classical example used to demonstrate the effectiveness of general algorithms. This is because the problem is simple and easy to understand, yet complex to solve efficiently while also being applicable in a surprising amount of applications such as neural networks and traffic control [BS09]. Relevantly for this thesis, it may also be interpreted as a Constraint Satisfaction Problem.

Because the N-Queens problem has been studied for such a long time, there have been a lot of different approaches to solve it. Traditionally and for small problems, the N-Queens problem has been stated as a CSP and solved by a Backtrack Search algorithm, as we describe in section 2.3. There are many other approaches that have been taken, such as genetic algorithms [HTA92]. Of importance for this thesis is that the N-Queens problem can be solved by using a strategy called Randomized Incremental Constructions as we describe in chapter 3.

This thesis is not meant to present the most efficient known ways to solve the N-Queens problem. Several approaches, that solve the N-Queens problem faster than the collaborative approach presented in this thesis, have been demonstrated before, such as [SG91]. Rather, the N-Queens problem is used in this thesis as an example application to demonstrate the efficacy of collaboration in parallel processing of randomized solution strategies as opposed to naive randomization without collaboration. This collaborative approach has very broad applications and is easy to customize for specific problems by providing simple operations. It is also well suited for massively parallel processing, making use of the trend of modern hardware to have increasing capabilities for concurrent execution.

Strategies for solving the N-Queens problem may try to find some, all or only one possible solution to the problem. In this thesis, we only consider approaches that generate one solution. The Randomized Incremental Constructions approach is based on randomization, which lends itself to search for a single solution. Furthermore, the count of all solutions for a given N is quickly too large to be counted [Slo].

2.2 N-Queens as a CSP

The N-Queens Problem can be stated as a CSP. There are many ways to specify the different aspects. In this thesis, the variables are the positions of each queen. The goal is to place N queens, so there are N variables $Q_1, ..., Q_N$. There are also three main constraints that apply to each variable, and need to be checked for each queen position. The first constraint is that no queen may be on the same row as another queen. We do not actually need to check this constraint if we incorporate it into the values that variables may take. If we say that each queen may only take a value in $\{1, ..., N\}$, and that the variable's index specifies its row, there can never be two or more queens on the same row. That way, the values of variables specify the column position of that queen. So $Q_5 = 6$ means that this queen is on column position 6 in row 5. Because of this, it is sufficient to save the column position in a one-dimensional array of integers with length N. By incorporating this constraint into the values, it is always true and does not need to be checked for any of the computations below.

The second constraint is that no queen may be on the same column as another queen. So each column position may only occur once between all variables. This is true if the values of variables are a permutation of $(1, ..., N)$. If we can ensure that at any point, the variable values are always a permutation of $(1, ..., N)$, we do not need to actively check this constraint either. We can easily ensure that this constraint is always true, by initializing the queen positions as $(1, ..., N)$, and then only ever changing their positions by swapping the values of two variables. So if we wanted to set the column position of the queen in row 4 to 5, and the queen in row 6 currently has the column position 5, we would need to swap $queens[4]$ and $queens[6]$. This is sufficient here because all approaches presented in this thesis operate incrementally, meaning they assign queens row-by-row. That way, we can freely swap the column position of the queen in the current row with any queen past that current row without having to worry about changing the result. Only the queens in rows before the current row are actually set, the rest are just possible assignments. The only constraint that actually needs to be actively checked then is whether any queen occupies the same diagonal as another queen.

The diagonal constraint can be checked efficiently because of the following observation: Let i be a row index, and j be a column index. The sum of these two indexes, $i + j$, is constant on any diagonal with positive slope and the difference of indexes, $i - j$, is constant on any diagonal with negative slope. The value of

sums on different diagonal lines with positive slope are different and the values of differences on different diagonal lines with negative slope are also different. These observations are presented in [SG90], although the authors seem to have mixed up the slope for the diagonals. Because of these observations, each diagonal line can be uniquely identified by a single number. For the N-Queens problem with an NxN board, there are 2*N-1 diagonal lines with positive slope and 2*N-1 diagonal lines with negative slope. Assuming that the row positions for all queens are saved in an array called *queens*, then for any diagonal with positive slope, the unique number is $i + queens[i]$, and $i - queens[i] + N - 1$ for any diagonal with negative slope. Adding N-1 guarantees that any index is non-negative, which allows us to keep track of occupied diagonals by using two arrays of Boolean values of size 2*N-1 each. When placing a new queen, checking whether it occupies a the same diagonal as a previous queen can be done in constant time by calculating its diagonal indices and checking whether those have already been set. This observation is used to quickly check the diagonal constraint in all implementations below.

2.3 Backtrack Search

A simple, general approach to solving CSPs is an algorithm called Backtrack Search. The goal of this algorithm is to build a valid solution incrementally by traversing the variables in a set order. At each step, the algorithm is taking a sub-sequence of provisional values $(X_1, ..., X_i)$ that satisfy all constraints and appending a new assignment of X_{i+1}, such that the whole new sub-sequence $(X_1, ..., X_{i+1})$ satisfies all the constraints applicable to it. In order to find which assignment may be appended to the sub-sequence, all possible assignments are traversed in a deterministic order and for each possible assignment, all constraints that affect this variable must be checked. If all constraints are satisfied for this variable, then it is added to the provisional sub-sequence and the process repeats. In case the possible assignment violates any constraints, the next value in the deterministic order is tried instead. There is a finite set of possible extensions, and if no value can be appended without violating any constraints, a dead-end situation occurs. In that case, the assignment for one of the prior variables must have been incorrect. The algorithm then backtracks to the previously assigned variable, attempts to change it by attempting to append another value and continues the process with the new assignment [DP87].

When searching for only one solution, the algorithm terminates if a sequence of N consistent values have been found, and returns that sequence. Alternatively, if no more possible assignments can be found for the first value, it also terminates and returns an empty sequence. For solving the N-Queens problem, Backtrack search would proceed as follows:

- Starting in the first row, place a queen at the leftmost position possible. By using the permutation trick for eliminating the column constraint, this would be iterating through all rows below this row, and swapping the queen position with the first row where the assignment would be valid after the swap. Save the diagonals that it can threaten.
- Move to the next row, and place a queen in a position so that it can not threaten the same column or diagonals as any previous queens. Also save the diagonals that this queen threatens.
- If no position in the next row can be picked without threatening any previous queen, go back to the last row and move the queen on that last row into the next available position in its row. This step is called *backtracking*.
- Terminate if a solution has been found that places one queen in each row, and return that solution. This always terminates for N > 3 because Backtrack Search exhaustively searches through all possible assignments and there is always a solution to the N-Queens problem for N > 3. A proof for the existence of a solution is shown on page 6 of [BS09].

The algorithm is called Backtrack Search because it can be interpreted as a search algorithm through a tree of possible queen placements. A partial tree of solutions and sub-solutions for the 4-Queens Problem is shown in Fig. 2.1. Each child of a node is a possible extension of that node. Orange tiles represent queens, and gray tiles show all threatened positions. A red X on a node without children means that there are no more possible extensions to that sub-sequence. The goal of search here is finding a node with depth 5, counting the root as depth 1. The simplest Backtrack Search would search depth-first through the leftmost possible child, backing up if a node has no more children and is not depth 5, then search through a different child. After finding a node of depth 5, the algorithm terminates, so no further nodes are depicted in the graphic.

2.3.1 Backtrack Search Implementation

Backtrack Search algorithms are usually implemented recursively, progressively building a solution by choosing extensions from a shrinking subset of possible values. Each recursive step keeps track of which positions have already been tried for the queen on it's row. The C++ implementation shown in Fig. 2.3 uses the heuristic for quickly checking the diagonal constraint and keeps the column constraint true by only swapping queens to take positions. The queens are initialized as (0,...,N-1), which is not shown here. A helper function is used to calculate the next possible extension. Because we only swap with the following rows to extend the sub-sequence, we do not need to try every possible column assignment for a given row, only those that have not been assigned yet. Those are to be found in rows below the current row. So we can simply iterate through all the following rows, keeping track of which row we last tried to swap from. To calculate whether an assignment is valid, we need to check if the diagonals that the queen would threaten after a swap are occupied . The diagonal numbers for that are $row + queen[j]$ and $row - queen[j] + N - 1$, where j is the row index of the queen that is to be swapped. The arrays that keep track which diagonals with positive slope and negative slope are occupied are called *dp* and *dn*, respectively. When an assignment is made, we set those diagonals as occupied and the next row is approached through a recursive call, ending when all rows have been assigned. If the next row has no possible extensions, meaning the recursive call returns false, we undo the diagonal occupation. Because the column constraint is always satisfied, it is not necessary to swap the queens back when undoing a placement. Both the queen in the row below and the queen on the current row enter the pool of non-assigned queens. Note that this algorithm has exponential worst case time complexity, and is not a feasible strategy for solving a CSP for big N. This simple backtracking algorithm is only used in practice for CSPs that have no such requirements, like solvers for simple games such as Sudoku. The time this algorithm takes to solve any specific problem varies wildly with how the board is initialized. When initializing it as (0,...,N-1), problems up to size 34 are solved in less than 20 seconds. However, the problem of size 35 is not solved, even after waiting many minutes. Improved versions of the Backtrack Search algorithm exist. They usually try to improve the selection of candidate assignments, such as [HE80]. However, a different strategy is preferred where possible.

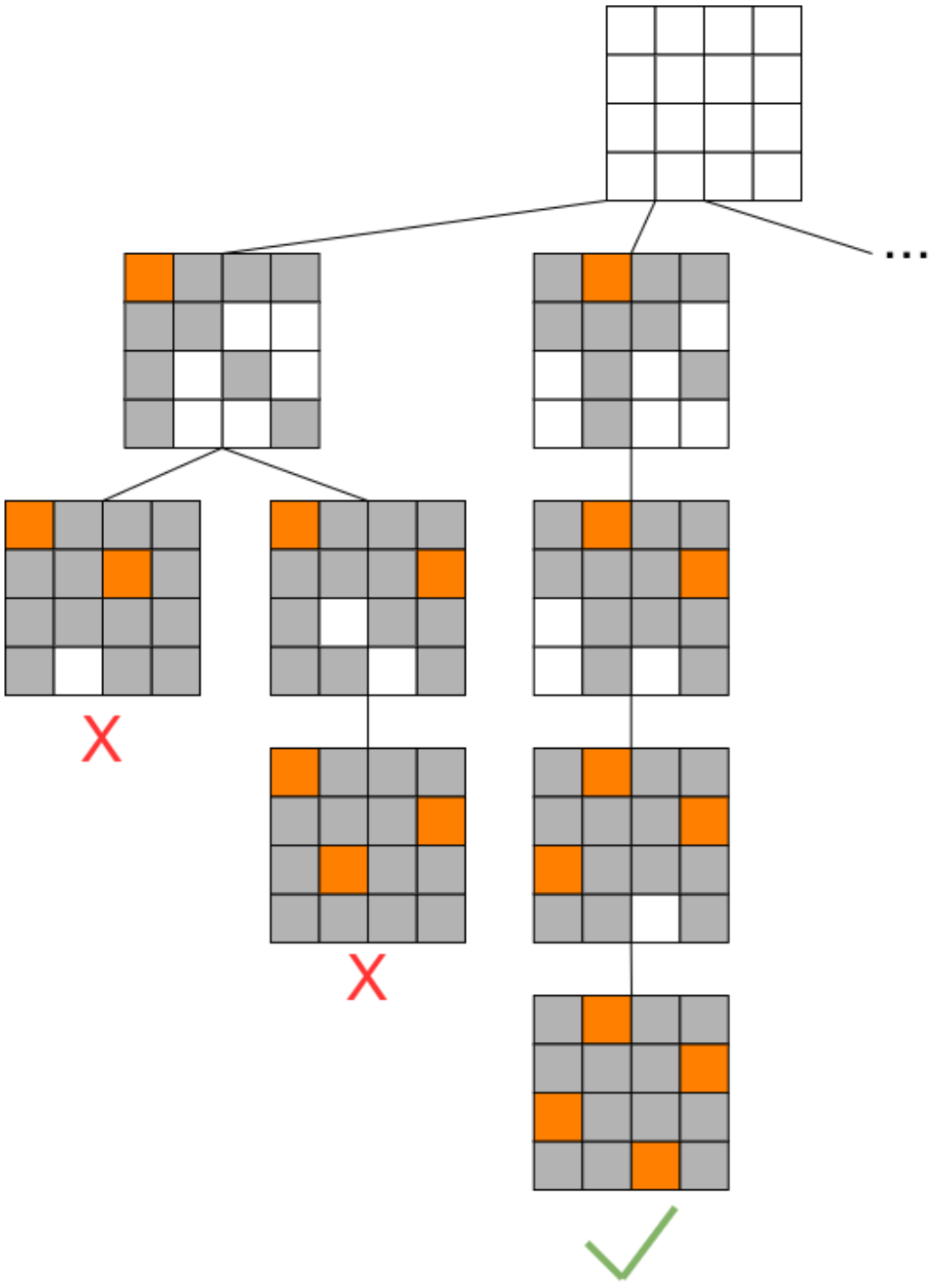

Figure 2.1: Tree structure of possible queen assignments to the 4-Queens Problem for Backtrack Search

```
//Calculate the next possible extension, starting from the last attempted extension
int next_extension(int row, int current_extension){
  for(int j = current_extension; j < N; ++j){
    int jp = row + queens[j];
    int jn = row - queens[j] + N-1;
  //Check diagonal constraints at the targeted position
    if(!dp[jp] && !dn[jn]){
      return j;
    }
  }
  //no extension is possible
  return N;
}
```

Figure 2.2: Helper function to find next extension for Backtrack Seach

```

bool backtrack_recursive(int row = 0){
  //Terminate if all queens have been placed
  if(row == N){
    return true;
  }

  current_extension = next_extension(row, row);
  //Try each possible extension in a deterministic order
  while(current_extension < N){
    //Place queen in column
    std::swap(queens[row], queens[current_extension]);

    //Set diagonals as occupied
    int ip = row + queens[row];
    int in = row - queens[row] + N-1;
    dp[ip] = true;
    dn[in] = true;

    //Extend next row
    if(backtrack_recursive(row+1)){
      return true;
    }
    else{
      //undo queen placement.
      dp[ip] = false;
      dn[in] = false;
    }
    //Go to next possible extension
    current_extension = next_extension(row, current_extension+1);
  }
  //No extension found
  return false;
}
```

Figure 2.3: Backtrack Search Implementation

3 Randomized Incremental Constructions

Instead of trying to search through every possible solution in a deterministic order, the following approach can also be used to solve CSPs. In fact, it has even greater applications, being useful for computational geometry, specifically of Delaunay and Voronoi diagrams [LS92]. It is also used for optimization in LP-type problems through Seidel's algorithm, originally presented in [Sei91] but later adapted for LP-type optimization problems. The approach is similar in structure to Backtrack Search, in that it also attempts to incrementally build a solution by extending a sub-solution N times with a possible extension, starting with an empty sequence. The crucial difference is that the choice of extension is not deterministic, but random. Instead of trying to improve a single partial solution by backtracking if the chosen branch of the tree has no more possible extensions, the Randomized Incremental Constructions (RIC) approach resets the sub-solution completely and begins the process again from an empty sequence. No attempt is made to improve any failed partial solution.
RIC can be stated generally for a huge class of problems, in terms of the following design pattern.

- Divide the solution to a problem into N discrete steps that can build on top of each other. This can be done by providing a data structure that represents the problem, on which the following operations can operate.
- Define an **increment** operation that extends a partial solution from i to $i + 1$ steps. This operation is crucial, because one extension has to be chosen from a set of possible extensions. For this, all possible extensions have to be computed first, and then the cornerstone of this design pattern is choosing an extension at random from all possible extensions. This operation returns true only if there is a viable extension that has been chosen and false if there is no possible extension.
- Define a **reset** operation that resets the partial solution if the increment operation can no longer extend the partial solution. This operation is almost always trivially easy to implement.

Fig. 3.1 shows the incremental and randomized nature of RIC. Each node represents a sub-solution. The number within that node specifies how many times it has been extended successfully, meaning how many variables have been assigned so far. The Nth node is the end-point and the algorithm terminates upon reaching it. The edges between nodes signify change in the sub-solution. Edges annotated with "Inc" mean that an increment operation has been called on the previous node to extend it. Edges annotated with "Reset" mean a reset operation has been called on the previous node, resetting its progress to an empty sequence. The central path of nodes and filled out edges is the series of random choices in the increment operation that produce a correct solution that reaches the Nth step without resetting. Any edges and nodes with dashed lines are alternative sub-solutions that are possible through a different random choice of extension in the increment operation. Different choices may lead to worse solutions that have to be reset eventually.

This design pattern has several advantages. It is just as general as Backtrack Search, while having none of its disadvantages. RIC does not need to keep track of any previous choices, it does not need to keep extensive stack frames for recursion and a single bad path taken does not immediately produce an exponential time increase. Somewhat counter-intuitively, the worst case of taking many bad choices in a row rarely occurs in practice, even though the solution is purely random without any improvements. This leads to much faster correct solutions. It is also simple to implement for a specific problem, only requiring the implementation of a data structure representation of the specific problem as well as the increment and reset operations for that specific problem. Most importantly for this thesis, it can also be run in parallel easily and effectively as shown in section 4.2. A general pseudo-code structure for the algorithm is shown in Algorithm 1. Note

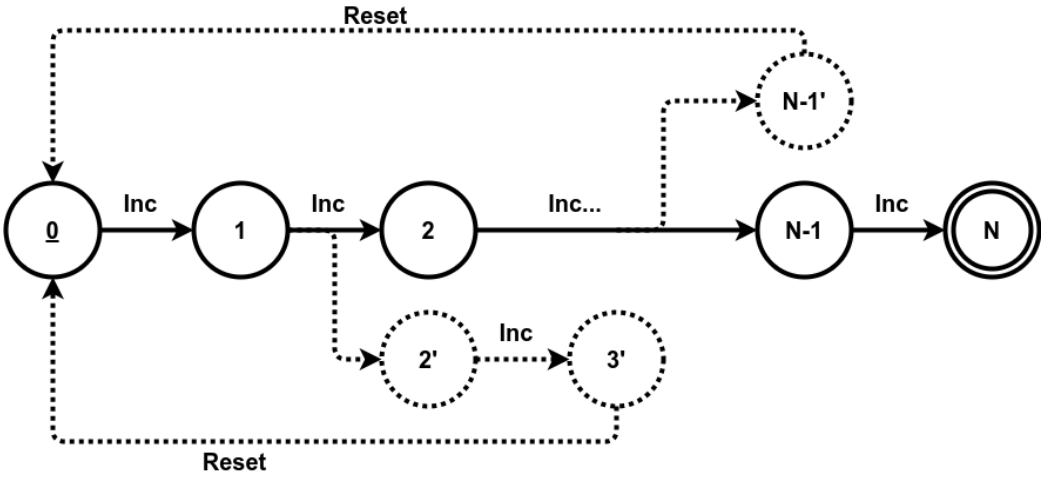

Figure 3.1: Incremental nature of RIC with divergent paths symbolizing random choices

that this relies on a solution being possible. As stated in section 2.3, a solution to the N-Queens problem always exists for N > 3. An empty solution is defined by the specific data structure used to represent the problem. We start with an empty solution, and then try to take N steps. In each step, the increment operation is called, which searches for any possible extensions and performs one at random if possible. In case the increment operation fails to extend, the whole solution is reset, and the process starts anew. If the increment operation is called N times without failure, the process finishes and returns the correct solution.

Algorithm 1 Randomized Incremental Construction Skeleton

```
1:  procedure RIC_SKELETON(N)
2:      success ← false
3:      solution ← empty solution
4:      while !success do
5:          success ← true
6:          for i ← 1 to N do
7:              if !increment(solution, N, i) then
8:                  reset(solution)
9:                  success ← false
10:                 break for loop
11:             end if
12:         end for
13:     end while
14:     return solution
15: end procedure
```

3.1 Randomized Incremental Constructions for N-Queens

When implementing RIC for a specific problem, we need to define a data structure, an increment and a reset operation. The data structure of a solution to the N-Queens problem is simple. For the pseudo-code operations shown below, we assume that the data structure has a member variable *queens* that is an array of integers with length N, holding the column positions of queens. As described in section 2.2, this is enough

to describe every possible position for a queen. The value of *solution.queens[i]* is the column position of the queen in row i. The data structure also has two member variables *dp* and *dn*, which are both arrays of length $2 * N - 1$ of Boolean values. Which diagonals with positive and negative slope are occupied is saved in dp and dn respectively. An empty solution as referenced in the general skeleton means that the queens array is (1,...,N) and the dp and dn arrays hold only false as their values.

The specific increment operation for N-Queens could be defined as in Algorithm 2. It calculates possible positions for the queen in the next row and randomly chooses one of those positions. i is the next row number. Like the Backtrack Search implementation, any positions are taken by swapping positions and the diagonal heuristic is used to quickly check the diagonal constraint. Calculating all possible extensions is done in a similar way to the helper function in Backtrack Search, by iterating through all rows below the current row and checking each diagonal if the swap into that position were to happen. In order to make choosing a random possible extension easier, all possible extensions are swapped into the rows immediately following the current row. Because none of those are set yet, this does not influence the sub-solution itself, only make the randomization process easier. Alternatively, we could set up a separate table for which indices can be swapped for extensions, and choose randomly from that. We decided against that, because a swap operation for the N-Queens problem is not much more expensive than a single assignment of an integer. In case that no possible extension was found, we return false. Otherwise, we chose a random row number of the possible extensions and swap column positions with the queen in that row and return true.

The specific reset operation could be similar to Algorithm 3. It simply resets the taken diagonals, since all other constraints are implicitly true when we only swap queens.

Put together, the N-Queens RIC algorithm iterates through the rows, placing one queen in a random possible position, then moving on to the next row and placing a queen at a random possible position in that row. In case a queen can not be placed in any position on the current row without threatening any previous queen, the whole attempt to solve the problem is discarded. If N queens have been placed successfully, the placements of those queens are returned.

Algorithm 2 N-Queens Increment Operation

```
1:  procedure N_Queens_INCREMENT(solution, N, i)        ▷ solution is to be passed by reference
2:      k ← i
3:      for j ← i to N do                               ▷ Swap any viable queens to the front
4:          jp ← i + solution.queens[j]                 ▷ Diagonal numbers after swap
5:          jn ← i − solution.queens[j] + N − 1
6:          if !solution.dp[jp] and !solution.dn[jn] then
7:              swap(solution.queens[k], solution.queens[j])
8:              k ← k + 1
9:          end if
10:     end for
11:     if k == i then                                  ▷ No choice can be made
12:         return false
13:     end if
14:     h ←random number in {i, ..., k}
15:     swap(solution.queens[h], solution.queens[i])
16:     solution.dp[i + solution.queens[i]] ← true      ▷ Set diagonals as occupied
17:     solution.dp[i − solution.queens[i] + N − 1] ← true
18:     return true
19: end procedure
```

Algorithm 3 N-Queens Reset Operation

```
1: procedure N_QUEENS_RESET(solution,N)
2:     for i ← 1 to 2 * N − 1 do
3:         solution.dp[i] ← false
4:         solution.dn[i] ← false
5:     end for
6: end procedure
```

3.2 A comparison of RIC and Backtrack Search

A direct comparison to Backtrack Search is hard to perform. Backtrack Search may take extreme amounts of time to find a solution, if the initial ordering of queens is bad but finds a solution quickly if the initial ordering is good. This is clearly shown by not terminating even after minutes of trying to solve the 35-Queens problem yet being able to almost instantly solve the 36-Queens problem.
For the comparison shown in Fig. 3.2, the initial ordering has been randomized and the time spent for each data point is taken as the mean value of 10000 random samples. The x-axis denotes the problem size (how many queens are to be placed), while the values show the average required steps to solve a problem of that size. A single-threaded version of the RIC implementation shown in Fig. 5.2 is used for comparison. The RIC values are also taken as the mean value of 10000 random samples. A step is going from one node in the tree of possible sub-solutions to the next. For Backtrack Search, this is a recursive call and for RIC, this is calling an increment operation. Even for these small problem sizes, Backtrack Search performs so much worse that testing it for any larger problems is almost impossible. It is clear that Backtrack Search may not be used in practice for anything other than very small problems. While RIC allows us to solve bigger problems, it would be preferable to speed up the execution of RIC even further.

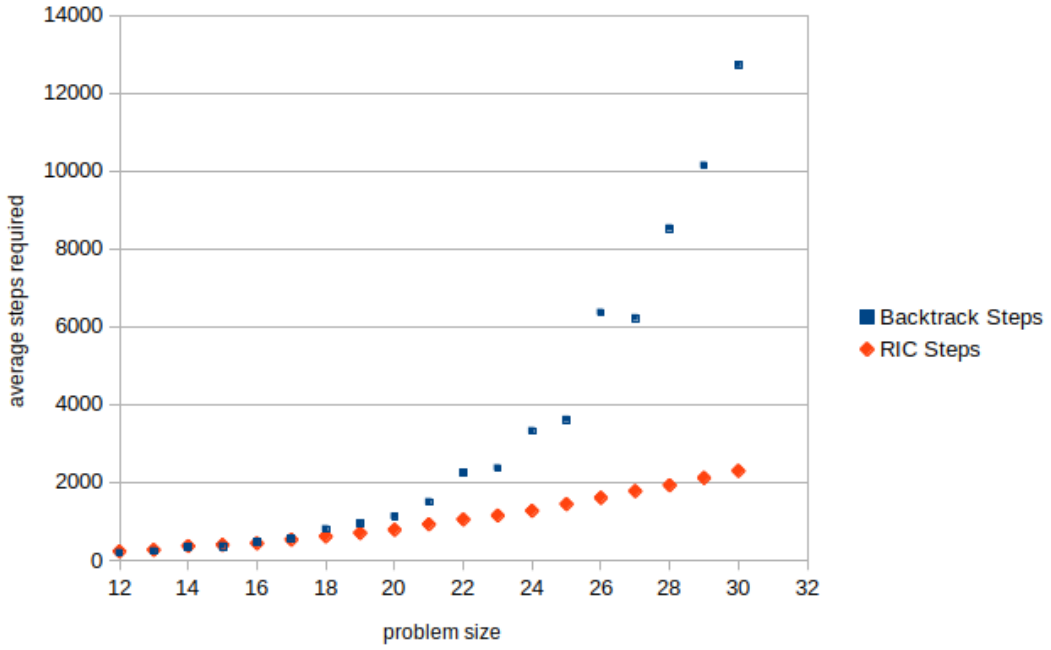

Figure 3.2: Comparing required step counts of Backtrack Search to single-threaded RIC. Lower is better

4 Parallel Computing

4.1 Challenges in High Performance Computing

Parallel computation is executing some task with multiple computing units at the same time. In recent years, parallel computation has gained increasing relevance. This is mainly due to high demand for increasingly large-scale applications as well as changes in computer architecture design that necessitate parallel algorithms to unlock their fullest potential. And even though the single-threaded RIC approach performs well compared to Backtrack Search, it still requires a long time to complete large problems. Fortunately, it is well suited to parallel execution.

These architectural changes to necessitate parallel algorithms have happened because sequential computation may not be scaled arbitrarily. Famously, Moore's Law states that the amount of transistors on a microchip doubles every two years [Tar19]. This is an observational law, which has held true for much longer than most people predicted. For this to be possible, the size and distance of said transistors has to be reduced. However, packing so many transistors close together brings problems that may not be solvable, like lack of heat dissipation and shortages in power supply [Per19]. Larger chips with multiple cores allow for better spread of these transistors, while still granting increased computing power when used to their full potential.

Furthermore, even though processing speeds might not increase as quickly in the future, memory access speed is increasing even more slowly. This is called the memory wall, a term popularized by [WM96]. The memory wall refers to the fear, that the processor/memory performance gap grows so wide, that almost all computation may be limited by memory latency and bandwidth. To combat this, increasing hierarchies of smaller and faster memory, called caches, have been instituted to decrease memory latency. However, effective caches are expensive. Extensive use of caches may also run into the same problem regardless, if the required data is too large to fit into cache, or too varied to be effectively cached. Parallel computing can reduce necessary memory access by reducing the amount of communication necessary, and increasing the load of local computation within a core. This trade-off needs to be addressed by designing parallel algorithms that have a significant amount of local load rather than excessive communication between threads.

Modern architectures have several degrees of parallelism. Interesting for us are the following.

- Modern CPU architectures try to avoid the end of Moore's Law by combining multiple cores into one processor. Each core can compute independently, and is able to act on different data and different tasks from the other cores. Cores typically share memory, so communication can be organized effectively through shared data. 'Modern CPUs have between two and 32 cores, with most processors containing four to eight. Each one is capable of handling its own tasks.' ([Saf20]). Because each core is independent, to fully saturate the possibilities of computation, we need to keep every core busy as much as possible. This is done by defining some number of logical tasks, called threads, that are assigned to the different cores by the operating system. So to split computation across threads, it is necessary to define sub-tasks that can operate concurrently with each other. For defining efficient algorithms, this is often done done by splitting the whole computation into a discrete number of sub-tasks. A classic example of this is matrix multiplication, where the result of a set number of matrix elements is calculated in each thread. A simple implementation for using all cores of a CPU using OpenMP is shown in [Alw17]. Because the given task has a discrete number of steps, there is a hard cap on how often it can be split up. So there is an inherent limit to how many threads may be started. Relevant for this thesis is another way of defining tasks for threads. The idea is to define a task that may be run many times

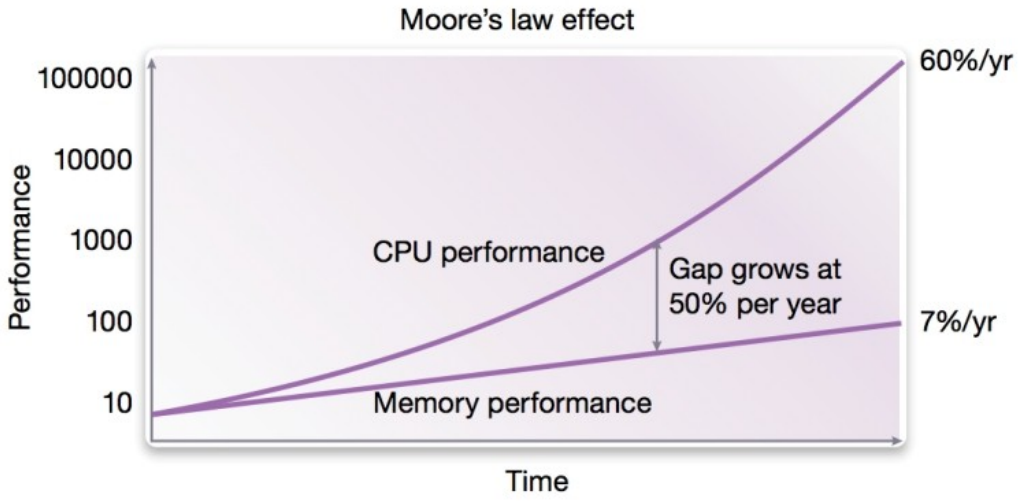

Figure 4.1: The increasing gap in CPU performance and memory access speed as reported in [Byr14]

concurrently as a whole while leading to some overall benefit. The actual workload of the task itself is not parallelized. If there is always an inherent benefit in starting more threads, regardless of how many have already been started, there is no hard limit on the number of threads we can start. There is currently a trend of hardware developers to include a lot more cores in their high-end CPUs, such as the AMD EPYC series, which may have up to 128 logical cores [Dev19]. There is also currently a research project by Intel called the Tera-scale project to develop architectures which involve a lot more cores, described in this inspiring but slightly outdated article [Rat12]. Defining tasks that can scale effectively to a large number of threads may be important to the future for CPU architecture development.

- Special instructions that can process a single instruction on multiple data points concurrently, called SIMD instructions or vectorized instructions [Erl]. They are operating within a single core of modern CPU architectures or within a group of threads on GPU architectures. Most notably, they allow to read and write more than one word size of continuous memory in one instruction. Because memory access speeds are a bottleneck on a lot of computation, and may be even more so in the future, algorithms designed to read and write as much continuous memory as possible can make use of these SIMD instructions to significantly increase memory bandwidth. It is also possible to do SIMD calculations, but that is not relevant for this thesis.

- Modern GPU architectures already use a large number of computing units, and comparatively little memory. They are used to speed up tasks that can be formulated as a huge amount of sub tasks. This is usually some matrix of independent calculations that need to be done. Traditionally, this is calculating the color values of each pixel on a screen. However, modern GPUs can be used for arbitrary computation. Specifically, they have a lot more computing units, and would allow us to scale our approaches even further. However, they are not as flexible as CPU cores, and there are a number of special limitations on writing programs for a GPU. The most important limitations are explained in section 7.1.

4.2 Running RIC in parallel

Even though the RIC approach is preferable to Backtrack search, it still struggles to find a fast solution to big problems when run sequentially. However, we are searching only for one solution. To speed up that search, we can execute several instances of the RIC algorithm described above concurrently, and terminate if any of them succeed. Doing this, we can reduce the total execution time to that of the fastest success in the threads. Because the choice of a candidate extension for a sub-solution is random, different threads choose distinct candidates, provided their pseudo-random number generation is seeded differently. This leads to varying execution times between threads. Fig. 4.2 shows four threads executing RIC in parallel. Each black box is an increment operation called on that threads solution. A red horizontal line signifies that the instance failed to find any possible extensions, and that a reset is necessary. It is possible to terminate after the second thread is finished. Continuation of the other threads is displayed only to show the possible variance in the number of steps.

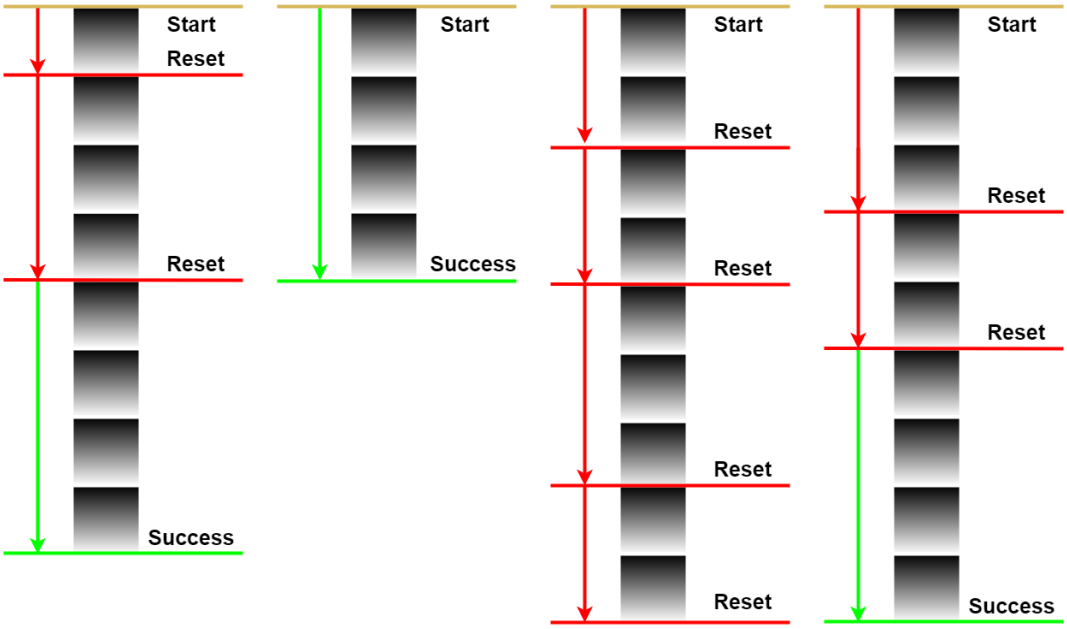

Figure 4.2: Parallel execution of RIC. Different lengths show divergence due to randomness

Let m be the number of instances running concurrently and let p be the average probability for success on any given step. Assuming the probabilities for each thread are independent, the probability that no thread succeeds in any given step is $(1-p)^m$. So the probability of achieving at least one success in any given step is $1-(1-p)^m$, which is always greater than p for $m > 0$, and $0 < p < 1$. So we have an inherent increase in the probability of termination for executing instances in parallel. This speedup has to be evaluated against any necessary communication and system overhead. Because instances calculate their own positions independently, communication is only necessary in order to terminate all instances when one solution has been found. This communication is minimal, and can be organized by using a single shared Boolean variable. Therefore, the communication cost is very small. A precise approximation for p and the cost of communication is beyond the scope of this thesis, and would depend on the specific problem. Furthermore, the communication cost is low compared to the time spend running in parallel for any problem. As shown in section 6.2, there is a significant speedup for any parallel execution compared to single-threaded execution.

In addition, this approach has almost no cap in the number of threads that can be executed. The communication cost does not scale at all with the amount of threads and increasing m always decreases $1-(1-p)^m$, so this approach can be scaled to any amount of threads given adequate hardware. The scalability of parallelism is capped only by how many random choices can be made. The amount of possible random choices scales exponentially with the problem size, so it is not a limiting factor on the thread count. Furthermore, this is not an issue because the number of possible different choices is much larger than the number of hardware threads available for any problem of interesting size. Some considerations have to be made for starting too many threads because there is an inherent overhead in starting a thread, and we need to allocate some memory for each thread to operate on. If we start more threads than the hardware supports, there is also a recurring cost to switching between threads. In practice, we found that choosing a reasonable thread count for a given problem size depends on the given hardware. For tests on a 128 core CPU, we found that increasing the thread count beyond 128 reduces the real time required by very little while tests on an 8 core CPU saw significant speedups for increasing the thread count to 64 or even higher.

4.3 Collaborative Parallelism

Any improved convergence in the simple RIC approach is exclusively due to the variance in the number of necessary steps between instances. It assumes that any sub-solution is equally likely to produce a correct solution in the end, no matter how many queens have already been placed.
The idea of our approach is that we assume instead, that solutions which have placed more queens already are also more likely to have at least one viable solution. If we start a high number of threads concurrently, it is likely that there is at least one possible solution to be found in these threads if the right extensions are taken. Instead of completely resetting any failed solution, we copy the current progress of promising solutions to other instances when they fail. In the next increment step, these solutions are identical but are likely to choose different random extensions. Doing this, we increase the amount of different random extensions that are explored on solutions that have already placed some queens successfully. Whenever any instance fails to extend its solution, it chooses a currently active instance and copies that instances progress to its own solution. Where to copy from is chosen at random from all currently active instances. Any extensions are chosen at random like above. A reset is performed only if all instances can no longer extend their sub-solutions. In that case, all instances are reset completely. This complete reset of all instances occurs far less than in the simple RIC approach. We call this approach *Collaborative Randomized Incremental Constructions*.

Fig. 4.3 shows this propagation of promising solutions between threads. A blue arrow indicates that a solution is copied to the instance that the arrow points to. After every horizontal red line indicating a failure extend the solution, another instance is copied from in order to proceed.

In order to implement our idea, we can reuse the increment and reset operations from above, and we only need so supply a copy operation, which is trivial in most cases. For N-Queens, this operation only copies over the arrays for the placed queens, the occupied diagonals and the next row that is to be placed. Due to its simple nature, it is possible that this specific approach or a similar one has been presented before, but we have not been able to find any published material that shows this approach under a different name.

Collaborative parallelism has a few interesting properties that make it an improvement over pure randomization, especially on modern hardware.

- When a solution is copied in case of failure, the current row of that solution is also copied and increased in the next increment operation. This leads to any one path down the search tree still being explored for a maximum of N steps. Because of this, collaborative RIC does not have the disadvantage of Backtrack Search that it may get stuck exploring bad paths in the tree for a long time while still exploring promising solutions more thoroughly than the naive RIC approach.
- We replace a reset operation in the scheme above with a copy operation for any threads that fail,

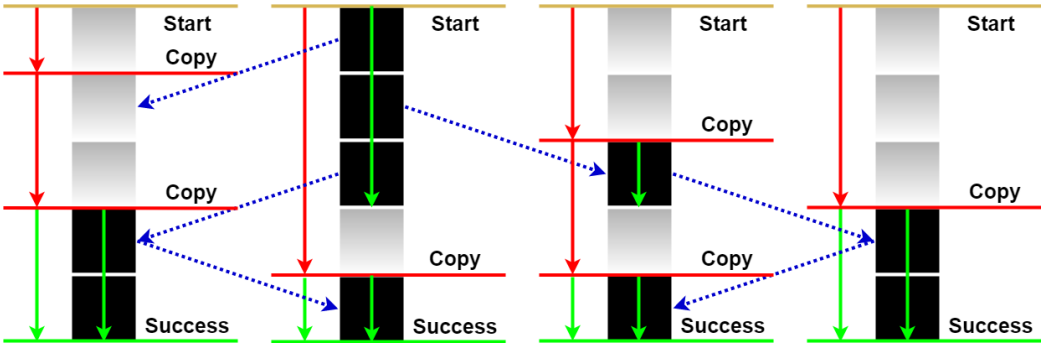

Figure 4.3: Collaborative solution sharing between threads propagates good solutions

assuming there are still any active solutions left. The probability that all threads fail at the same time decreases with the number of threads that get started. In this way, we can trade processing speed for memory usage. An increase in started threads increases the necessary memory, but decreases the chance of a reset being required. This flexibility is useful for adapting the collaborative approach to different architectures.

- Because we copy whole solutions, and those solutions can be stored in continuous memory, the memory access necessary can be done by vectorized (SIMD) instructions. Modern CPUs and GPUs increasingly make use of SIMD instructions for reading and writing to memory, to avoid comparatively slow memory access. Making use of these instructions can increase processing speed considerably.

5 CPU Implementations

All implementations for the following benchmarks have been done in C++. The RIC approaches for both the naive and the collaborative versions have been implemented by class `Solution` with an interface as displayed in Fig. 5.1. The integer N signifies the problem size, meaning a problem for an NxN chess board is to be solved. The integer i signifies the next row a queen is to be placed on in the increment operation. The class holds two member arrays of type `bool` with size 2*N-1 for the diagonals and an array of integers with size N for the queen positions. All memory is dynamically allocated so we can easily compare Solutions of various sizes through benchmarks. The class also has member functions for the increment, reset and copy operations. The implementations of the *increment* and *reset* functions are almost identical to the pseudo-code structure displayed in section 3.1 and thus are not shown here. The copy operation is implemented as a copy constructor.

To run different instances of these `Solutions` in parallel, a simple wrapper class `Instance` has been created, which holds a `Solution` and a Boolean value indicating whether it is still *active*, as well as an RNG object to make random choices. The random number engine is seeded in a unique manner for each thread and sample. Every time an increment operation is called, a global steps counter is incremented atomically. Counting the total number of steps allows us to compare the approaches in a hardware-independent manner and prevent interference from other outside factors.

5.1 Naive RIC

For CPU parallelisation, we use the OpenMP API [Boa] to execute instances in parallel. For that, an array with an `Instance` for each thread is created and then executed as shown in Fig. 5.2. A parallel block with a shared variable *success* is executed *thread_count* times. The parallel block is similar to the pseudo-code shown before. Each instance tries to extend the solution by calling the increment operation, resetting if it cannot and setting the shared variable *success* to true if it extended N times without failure. Before each step, all threads check whether another thread has already succeeded and cancel their execution early if any did. At that point, all threads stop execution before their next step. Note that for testing purposes, the actual correct solution is not returned. It could be returned through an out-parameter when setting success to true.

```
class Solution{
public:
  int N;
  int* queens;
  bool* dp;
  bool* dn;
  int i = 0;

  Solution(const Solution& s){
    N = s.N;
    queens = new int[N];
    dp = new bool[2*N-1];
    dn = new bool[2*N-1];

    for(int j = 0; j < N; ++j){
      queens[j] = s.queens[j];
    }
    for(int j = 0; j < N; ++j){
      dp[j] = s.dp[j];
      dn[j] = s.dn[j];
    }

    i = s.i;
  }

  Solution(int n){
  // Initialize queens to (0,...,n-1)
  }

  ~Solution(){
    delete[] queens;
    delete[] dp;
    delete[] dn;
  }

  bool increment(RNG rng){
  //Like previous pseudocode except that i gets incremented on success
  }
  void reset(){
  //Like previous pseudocode except i gets reset
  }

};
```

Figure 5.1: Solution class interface and copy constructor implementation

```
void simple_RIC(){
  omp_set_num_threads(thread_count);
  bool success = false;

  #pragma omp parallel shared(success)
  {
    int id = omp_get_thread_num();
    bool extended = true;
    instance[id]->reset();
    while(!success){
      while(instance[id]->current_row() < N){
        if (success){
          break;
        }

        extended = instance[id]->increment();
        if (!extended){
          instance[id]->reset();
          break;
        }
      }

      if(extended){
        success = true;
      }
    }
  }
}
```

Figure 5.2: Naive parallel RIC implementation in C++

5.2 Collaborative RIC

The collaborative approach is similar to the naive RIC approach, but instead of resetting on failure to extend, the threads attempt to copy from other active threads. If there is no thread that can be copied from, the instance resets. When copying from one instance to another, we need to make sure the copied instance does not get changed by a reset or increment operation during the copy operation. For the CPU implementation, this can be done efficiently with a lock for each instance. A lock is a variable that must be set by a thread in order to access a block of code. If the lock has already been set by another thread, the second thread must wait until the lock is reset before continuing execution. This lets us define operations that may not be executed simultaneously, like *copy* and *increment* or *reset*. For this, each instance holds an `omp_lock_t` variable called *copy_lock*, and has a member function for copying from another instance that sets that instance's *copy_lock*. Each instance locks its own *copy_lock* when performing *reset* and *increment* operations. That way, no instance can be copied from while it changes its state. The *increment* operation wrapper in Fig. 5.3 uses simple member functions of the Instance class to set and reset this lock, increments a global step variable that is used in the experiments to compare the required work of different implementations, and keeps track of whether the solution in the Instance is still active after the increment operation. Likewise, the *reset* operation wrapper in Fig. 5.4 also sets and resets the lock, as well as setting active to true after a reset.

```
bool Instance::increment()
{
  set_copy_lock();

  #pragma omp atomic
  ++steps;

  active &= solution->increment(rng);

  reset_copy_lock();

 return active;
}
```

Figure 5.3: Lock-based increment wrapper with activity tracking and step counting

```
void Instance::reset(){
  set_copy_lock();

  solution->reset();
  active = true;

  reset_copy_lock();
}
```

Figure 5.4: Lock-based reset wrapper with activity tracking

As in the naive RIC implementation, the actual collaborative calculations are executed in a parallel block as shown in the code sample in Fig. 5.5. At the start, we set up an array of integers with length *thread_count* as (0,...,*thread_count*-1), which are the indices of threads. This is used in the *copy_from_active* operation to select which thread to copy from. Execution terminates if the shared variable *success* is true because another thread found a valid solution, or if the current thread extends a solution N times without failing its increment operation. However, in case the increment operation failed, a copy operation is called, and we only reset if that copy operation also fails.

```
omp_set_num_threads(thread_count);
bool success = false;
#pragma omp parallel shared(success)
{
  int* indices = new int[thread_count];
  for(int k = 0; k < thread_count; ++k){
    indices[k] = k;
  }
  int id = omp_get_thread_num();

  while(!success){
    instance[id]->reset();
    while (instance[id]->current_row() < N){
      //Terminate early if another thread has succeeded
      if(success){
        break;
      }
      //If we can't increment or copy, we need to reset
      if (!instance[id]->increment() && !copy_from_active(indices)){
        break;
      }
    }
    //Terminate if the solution has been extended N times
    if(instance[id]->current_row() == N){
      success = true;
    }
  }
  delete[] indices;
}
```

Figure 5.5: Parallel block of collaborative RIC implementation

For choosing an instance to copy, we use a modified version of Fisher-Yates Algorithm [Bla] for generating a random permutation of a collection. The Fisher-Yates algorithm operates by dividing the collection into two sub-collections, one for elements with finalized locations and one for elements with non-finalized locations. The algorithm works by repeatedly choosing a random index out of the non-finalized locations, and swapping the element at that index with the first element past the end of the non-finalized sub-collection. Then, the location of that swapped element is finalized, meaning the size of the sub-collection of finalized elements increases by one. This leads to the number of non-finalized elements decreasing by one in each step until the locations of all elements have been finalized. Because each element is swapped to the back when selected, each element is selected exactly once and the algorithm terminates after exactly M steps if the size of the collection is M.

For the code sample in Fig. 5.6 we modified the Fisher-Yates algorithm to iterate through all instances in random order to check if that instance is active. In order to prevent that every failed thread copies from the same active instance, we want choose a random active instance to copy from. We do this by generating a random permutation of the indices of threads. We could use the Fisher-Yates Algorithm to first create such a random permutation and then iterate over it. However, instead of that, we do our calculations while generating the permutation. The Fisher-Yates algorithm selects each element in the collection exactly once, operates in a randomized order and in linear time. Thus, the algorithm itself can also be used for randomized iteration over a collection. We modified this algorithm to check whether the instance at the randomly selected index has an active solution.
This is done in a helper operation as shown in Fig. 5.7. Because this operation must make sure that the other solution is not changed while being copied from, it also sets that instance's *copy_lock*. If the instance has an active solution, we copy that solution and interrupt the algorithm by returning true. If it does not, we swap this index to the back like the Fisher-Yates algorithm dictates. That way, this index will not be selected again.

If we iterate through all instances without finding an active instance to copy from, we return false. Because we only swap the indices after every step, this index array is always a permutation of (0,...,*thread_count*-1). Thus, it may be reused for the same operation in the future.
We use a separate array of instance indices instead of changing the order of the instances themselves because the instance array is used by multiple threads simultaneously. If we were to change the order of the instance array, any other thread that currently chooses an instance to copy from could potentially select the same instance multiple times or copy incorrect data.

Note that the first instance that fails to extend after another instance has reset will find that newly reset instance as the only active instance. This is because an instance only resets after checking all other instances and finding that none of them are active. The newly failed instance will then copy this freshly reset instance, which is equivalent to it also resetting. This will spread through all instances in the next step. This way, we can reset all instances without having to check whether all instances actually failed.

```
bool Instance::copy_from_active(int* indices){
  //Go through threads in random order and copy solution if it is active
  for(int x = thread_count-1; x >= 0; --x){
    int y = rng.random() % (x+1);
    if(instance[id]->copy_from_instance_if_active(instance[indices[y]])){
      return true;
    }
    else{
      //Swap index to the end
      std::swap(indices[x], indices[y]);
    }
  }
  //If no instance to copy from was found, go to reset of instance
  return false;
}
```

Figure 5.6: Operation for finding an active instance to copy from

```
bool Instance::copy_from_instance_if_active(Instance* instance){
  instance->set_copy_lock();

  if(instance->active){
    delete solution;
    solution = new Solution(*instance->solution);
    active = true;

    instance->reset_copy_lock();
    return true;
  }

  instance->reset_copy_lock();
  return false;
}
```

Figure 5.7: Lock-based Copy wrapper with activity tracking

6 Experimental

6.1 Scaling the problem size

Our first series of tests are aimed at comparing how well the naive RIC and the collaborative approach scale when solving problems of different sizes. For this, we conduct two trials.
The first trial is counting the total amount of steps (increment operations called) required to solve a problem of a given size with a given thread count. This allows us to see differences in the scaling behavior of the approaches, regardless of hardware-specifics. Most importantly, these tests are not influenced by interruption of the program due to other processes or any other reason that might temporarily slow down the execution of any test. Because both approaches are randomized, each data point is calculated by repeating the experiment a certain number of times, and taking the value as the mean step count of these samples. Problems with a size below 200 queens are sampled 500 times. To speed up the testing process, any data points above 200 are only sampled 250 times.
Counting steps is useful for receiving uninfluenced results, but in order to make sure that this is also a good measure for comparing the approaches in general, we conduct a second trial to see the required real time for solving problems of the same size. By presenting similar results, we show that counting steps is a good measure for performance.

6.1.1 Counting steps

The results for counting the required amount of steps are displayed in Fig. 6.1. This test is conducted on an AMD EPYC 7501 CPU that has 128 logical cores. Plotted on the x-axis are various problem sizes from 10 to 600. Plotted on the y-axis are the average step counts calculated as $\frac{total\_steps}{instance\_count * sample\_count}$. Dividing by the instance count allows us to see a speedup due to parallel execution and compare the effects of different thread counts.

The three upper curves in red, green and blue correspond to the naive version with 32, 64 and 128 threads, respectively. A smaller thread count leads to a higher necessary average step count. Specifically, the amount of required steps increases far more quickly for the 32-threaded naive version, due to a worse convergence on a correct solution with less parallelism. Because of this, we discontinue it after solving problem sizes of 200, because it would take too long to test for bigger problems.
We observe that increasing the thread count does not seem to reduce the overall complexity, but it does lead to a lot more shallow curves. Both of the other naive curves show much worse scaling behavior than the collaborative approaches. Even with 128 threads, the naive version still requires more average steps than the 32-threaded collaborative version, which is the almost hidden curve in gold. Both of the lowest curves in blue and violet correspond to the collaborative approach with 64 and 128 threads respectively. Both curves are performing much better for all data points past problem sizes of about 100. The difference between the collaborative curves is not as large as that of the naive ones. This chart is somewhat misleading in this regard though, because both collaborative approaches do not require nearly as many steps as the naive versions so their difference appears smaller in comparison.

Most surprising is the fact, that the slope of the collaborative curves does not seem to change much with increasing the problem size. Both curves seem to increase their slope so slowly that they almost appear to be linear, even up to problems of size 600. This is very surprising, as we did not expect an apparent change in the complexity class when solving collaboratively. Even in this, there is no true change in complexity, as the

slope still increases, albeit slowly. We expected similar curves as the naive version, which are simply more shallow, but not to this degree. Most likely, this behavior is not a true characteristic of the collaborative approach. It is rather likely that this apparently very slow scaling is merely the beginning of a shallow curve of higher order. It may also be some specific feature of the N-Queens problem itself that leads to this behavior.

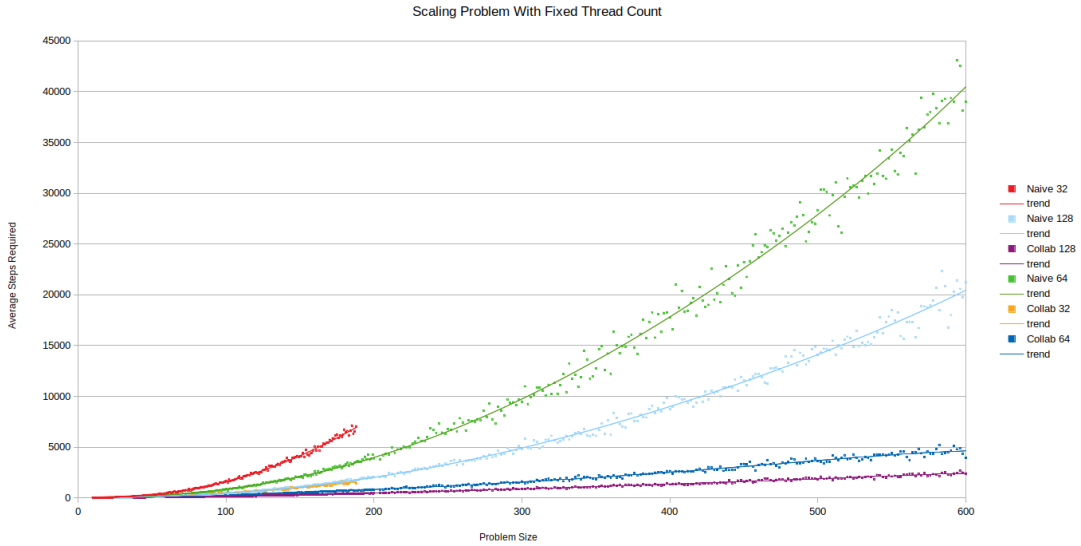

Figure 6.1: Comparison of scaling behavior in regard to the average step count for varying problem sizes with fixed thread counts. Lower is better

6.1.2 Counting real-time

Testing only for the amount of required steps ignores the inherent overhead of starting and managing more threads. Calculating the step count for each sample as the average of a fixed thread count also gives higher thread counts an inherent advantage, because each started thread influences the average result by the same amount, even if they may not take as many steps as other threads.

The chart displayed in Fig. 6.2 is identical to the prior chart in methodology and presentation, except that we count the real-time elapsed in seconds between starting the program and finishing it. This includes any required time for setup, starting and shutting down threads and any other overhead. We did not make any attempt to reduce the overhead for subsequent samples after the first. The plotted values on the y-axis are $\frac{total\_time\_elapsed}{sample\_count}$ for a given problem size.
We observe that a notable change in the behavior of all the curves is that they are closer together. This is to be expected, because there is a constant overhead for starting each thread, which is higher for those curves with a higher thread count. This counteracts some of the speedup due to a faster convergence. There is also some more constant overhead for starting any program and setup that does not depend on the problem size or thread count. So the actual variable execution time is only some percentage of the total execution time. This leads to more shallow curves for any solutions, which is especially visible for the naive RIC curves.

Due to these factors, the collaborative curves are even closer together, visibly separating only after a problem size of 450. Looking very closely, we can see that the 128 thread version actually has slightly higher execution times for problems of a size below 400, while having a lower average execution time for big

problems.

This suggests that the trade-of between a higher thread count and the problem size needs to considered actively when solving real problems. The naive version, while performing worse in general, does not seem to show this behavior. For the naive version, more threads clearly perform better in almost every situation. Note that real-time benchmarks are sensitive to many forms of interference, so specific data points may be slightly incorrect. Nevertheless, the real-time scaling behavior is similar to the step count scaling behavior. We conclude that counting steps is a good measure of performance.

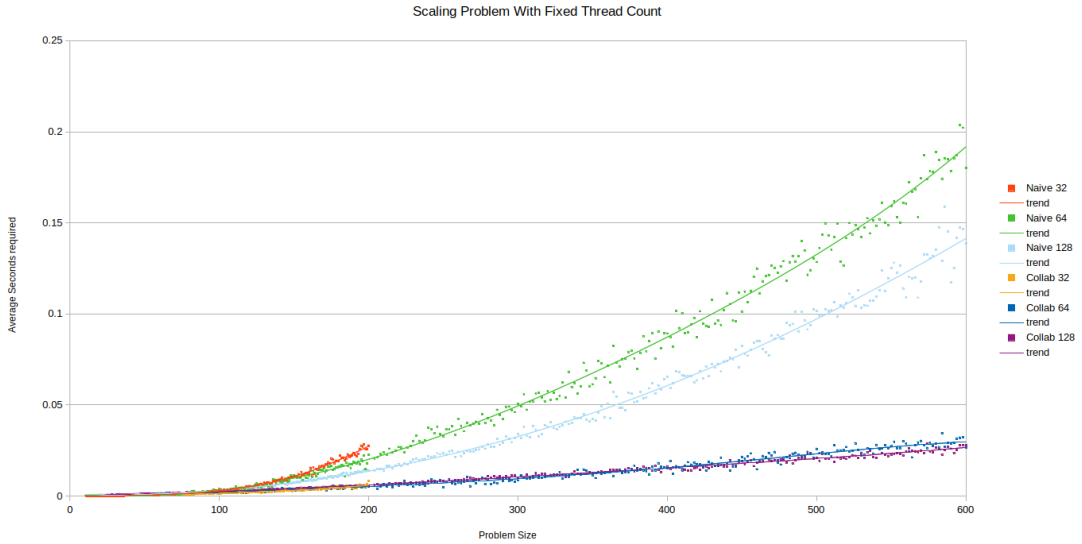

Figure 6.2: Comparison of real-time scaling behavior for varying problem sizes. Lower is better

6.2 Scaling the thread count

Our next test is aimed at figuring out the relative advantage of increasing the amount of parallelism in finding faster solutions for both the naive and collaborative RIC versions. For this, we execute fixed problem sizes with scaling thread counts, and we calculate a relative speedup factor as compared to the base case.

The results are displayed in Fig. 6.3. Plotted on the x-axis are various thread counts ranging from 4 to 256. Each data point is the average step count of a number of random samples. For 4 to 128, each point is sampled 250 times. For any data point above that, it is sampled 500 times. The y-axis denotes a relative speedup factor as compared to the execution on 4 threads. The value of a point x are calculated as $\frac{steps\_for\_4\_threads - steps\_for\_x\_threads}{steps\_for\_x\_threads}$. [1]
Both lower curves in green and violet show the speedup of the naive version for 200 and 400 queens respectively. We observe that both curves are linear, which is to be expected because this approach has no inherent advantage for increased parallelism except from a faster convergence. Especially due to the way we calculate steps as an average of the instance count, it is clear that these curves should be linear.

Surprisingly, the naive version for 400 queens seems to show some slightly higher speedup values compared to 200 queens, which might indicate that naive version performs especially poorly in low-threaded

[1] Taking 4 threads as the base comparison is an unfortunate choice. Due to the current crisis of the Covid-19 virus, we were unable to repeat this test for better values, as these tests are conducted on a machine that is provided by the currently inaccessible university.

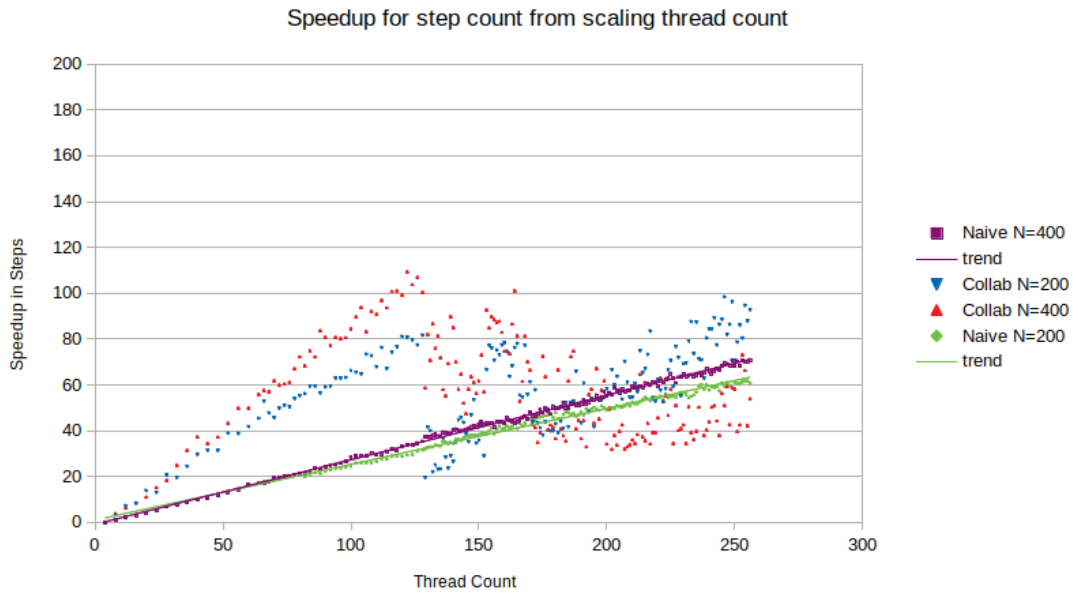

Figure 6.3: Comparison of speedup due to increased parallelism for 200 and 400 queens. Higher is better

execution for big problem sizes.
Both of the upper curves in blue and red correspond to the speedup for collaborative execution with 200 and 400 queens respectively. Both curves show a much better scaling behavior than their naive counterparts for 4 to 128 threads, achieving around 3-4x more speedup. Comparing only the part up to 128 threads, the collaborative approach seems to provide a significant improvement over the naive version, and increasing the amount of parallelism always brings an almost constant increase in the achieved speedup. This is consistent with the earlier tests with scaling problem sizes. This is desirable, as it indicates that more parallelism is inherently better for the collaborative version than the naive approach.

We were hoping that this behavior would show, albeit in a diminished form, even for higher thread counts. However, starting at exactly 129 threads, the speedup for the collaborative approach seems to break apart completely. Both collaborative curves increase a lot in variance, but they both show a similar pattern of decreased speedup for 129 to 155 threads, then an increase for 155 to 170 threads followed by another dip and a slowly increasing speedup for the remaining values. Interestingly, the curves also swapped in their relative speedup, with the 200 queens version achieving higher speedups than the 400 queens version for 200 or more threads. These patterns are highly surprising, and also very suspicious in their exact timing. The CPU we tested on has exactly 128 logical cores, and execution slows down significantly for 129 or more threads. Furthermore, the step count should not be dependent on any outside influence such as program interruption or scheduling changes. For the speedup to decrease for higher thread counts, and have such a strange pattern, there must be something wrong with our implementation. However, we found no obvious errors that would cause such behavior, and are unsure in general what could possibly cause such behavior. Our first hope was that this might be an error that is caused by using that specific CPU for computations, as the break-down occurs at exactly 129 threads.

However, testing on another machine shows similar behavior. The test displayed in Fig. 6.4 is conducted on an Intel Xeon CPU E5-1620 v2 with 8 logical cores. The chart shows only the scaling behavior of the collaborative approach for a problem with 200 queens. While this curve also show a similar break-down of the achieved speedup, with a subsequent increase, the results show a clearer pattern this time. There

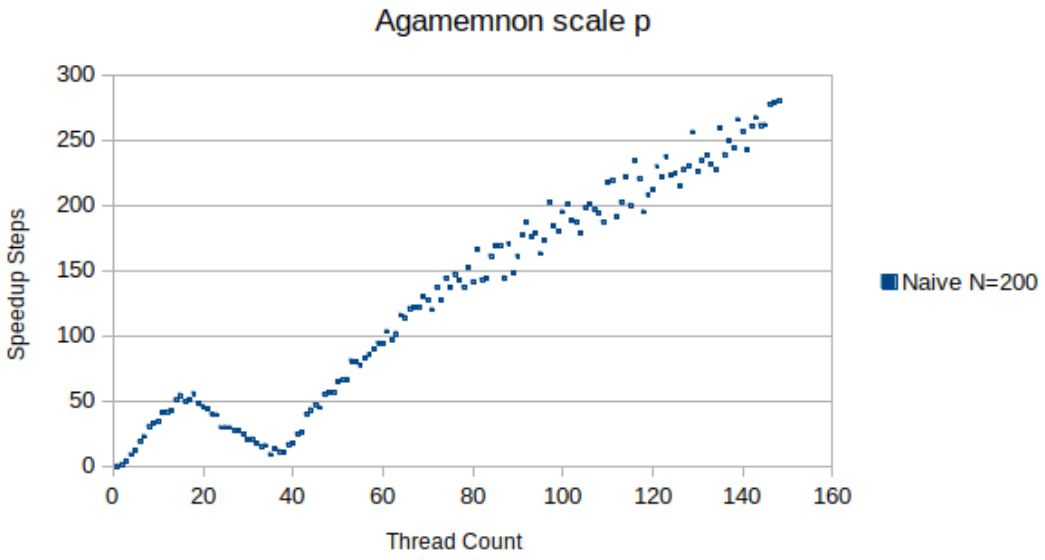

Figure 6.4: Speedup for increased parallelism of collaborative execution on a problem size of 200 on a machine with 8 logical cores.

is not nearly as much spread of nearby points, and no repeated increase or decrease but only a single decrease. Notably, the break-down is not as rapid as on the other machine, rather decreasing gradually before increasing again. The break-down is also not starting at 9 threads, as would be expected if we compare to the other machine, but at around 18. This indicates that the problem is more complicated than a defect machine or a break-down for a thread count higher than can be serviced without switching threads by the machine.

We have not been able to find an error in our implementation, and we see no inherent reason why a lock based implementation would have this break-down. In fact, after the break-down, the speedup increases to high values far surpassing the values before the break-down, which indicates good scaling even far beyond hardware capabilities. This increase leads us to believe that the problem is not due to starting too many threads, or using too many locks. Furthermore, the problem exists even when counting steps, not just when testing real-time performance. This indicates that some number of instances are not contributing to finding a correct solution, while still taking steps. The most likely situation where this would occur is when two instances copy from the same active instance and one of them is then unassigned from a core. If the other instance that is still actively computed then fails its increment operation and copies from that recently unassigned instance, it would copy the same solution again, failing again in the next step. However, for this to happen, it would need to randomly choose the same instance to copy from multiple times in a row. Furthermore, the instance to copy from must not be assigned to any core, otherwise it would simply do a step and copy from some other instance. Copying from the same instance multiple times in a row is more likely for low instance counts, and there must be more instances than logical cores in the CPU, in order for any thread to be unassigned. So this circular effect would be most pronounced for thread counts that just exceed the logical core count of the CPU, which lines up with our observations. We think it is unlikely that this is the sole reason for this break-down, especially on the 128-core machine. Increasing the thread-count from 128 to 129 caused the worst break-down on that machine, and the probability for selecting a single inactive thread over and over is too low for this to be the whole reason. It is possible that some characteristic of the OpenMP implementation, our pseudo-random number generation or the operating system scheduling leads to these circular copies occurring more frequently for those specific thread counts.

The performance for low and high thread counts is still very good, which makes this implementation useful for solving most problems. However, we would prefer to have an implementation that does not have this weakness of a broken speedup for some thread counts, so we conduce further experiments described in the following sections.

6.3 Barrier Implementation

In order to test further scaling of thread counts while avoiding the previously mentioned issues, we implement the collaborative approach without locks. However, letting threads run independently is also not an option. It would mean that some threads might try to extend their solution, while others try to copy from that solution, which breaks the copied solution. To combat this, we have to limit the freedom of threads to run independently. This can be done by synchronizing threads with a barrier, which all threads have to reach before continuing their computation. Thus, we split the execution into three phases and use barriers to separate them.

In the first phase, all threads attempt to extend their solutions and save whether they succeeded. In the second phase, we check whether all threads have failed their increment operations. This can be done by letting each thread set a shared variable *failAll* to false if the thread succeeded. This can also be done independently by each thread. The last phase is a choice between two paths for all threads to execute. If all threads failed previously, we reset all threads, which can be done independently. If at least one thread did not fail, all failed threads copy from other instances. After the third phase, all threads continue to the first phase, or terminate if any any threads extended N times. Because of the forced synchronization, all threads are always on the same step of extension, so almost all threads succeed at the same time.

The code fragment in Fig. 6.5 shows only the while loop executed by each thread. Note that we can discard the result of *copy_from_active* because we already know from the second phase that at least one thread is active. The *#omp single* directive has an implicit barrier following it, so we don't need to explicitly put a barrier there. Note that this implementation has a far worse real-time performance than the lock-based implementation for thread counts before the speedup break-down. In most cases, it is still preferable to execute the lock-based version. However, this implementation allows us to see if there is an inherent advantage of increasing the thread count to values far surpassing the hardware capabilities.

6.3.1 Testing Speedup for Barrier Implementation

All the following tests are conducted on an Intel Xeon Processor E5-1620 v2 with 8 logical cores. [2] Any speedup values displayed are not comparable to the speedup values of tests conducted on the better machine. We only hope to see a continuous increase of speedup values for higher thread counts. We calculate the speedup value for 1 to 256 threads. Each speedup value is relative to the single-threaded case and is sampled 500 times.

Fig. 6.6 shows the results for 200 and 400 queens. The results are similar to the earlier results, except that they show no break-down of speedup. We can also see some interesting effects due to scaling up to 256 threads. Notably, the lower curve for N=200 has a slope that decreases faster than the upper curve, which indicates that speedup values keep increasing for larger thread counts if the problem is larger. This leads to an increasing gap between the curves, meaning the collaborative approach is particularly well fitted for solving bigger problems, as the increased speedup scaling counteracts some of the extra problem difficulty.

However, this particular barrier-based implementation runs significantly slower in real time than a lock based implementation. This is to be expected, because each thread has to reach each barrier before continuing. Specifically for thread counts beyond the hardware capabilities, this requires a context switch for each thread

[2] Because of the current Covid-19 crisis, we can not repeat any tests on the 128 threaded machine.

```
while(failAll){
  instance[id]->reset();
  for (int i = 0 ; i < N ; ++i){
    extended = instance[id]->increment();
    #pragma omp single
    failAll = true;


    if(extended){
      failAll = false;
    }

    #pragma omp barrier

    if (!failAll){
      if (!extended){
        //We know that at least one instance is active here
        copy_from_active()
      }
    #pragma omp barrier
    }
    else{ //If all threads failed, reset them
      break;
    }
  }
}
```

Figure 6.5: Main loop of collaborative barrier implementation that is executed by each thread

which leads to lower CPU core occupancy. On this particular machine, each thread was active as little as 20% of the time for the highest thread counts. Especially on machines with few physical cores, this effect leads to a significantly slower real-time execution. Nevertheless, we can conclude that there is a continuous advantage for increased parallelism inherent to the collaborative approach, even far beyond the hardware capabilities. There is also no inherent reason why a break-down of speedup would occur, so we still see no reason to believe a correct lock-based implementation would be impossible. Considering that the scaling behavior for the lock-based implementation is very good after the break-down, and we showed that the approach has inherent scalability, a fast and well-scaling implementation should be possible.

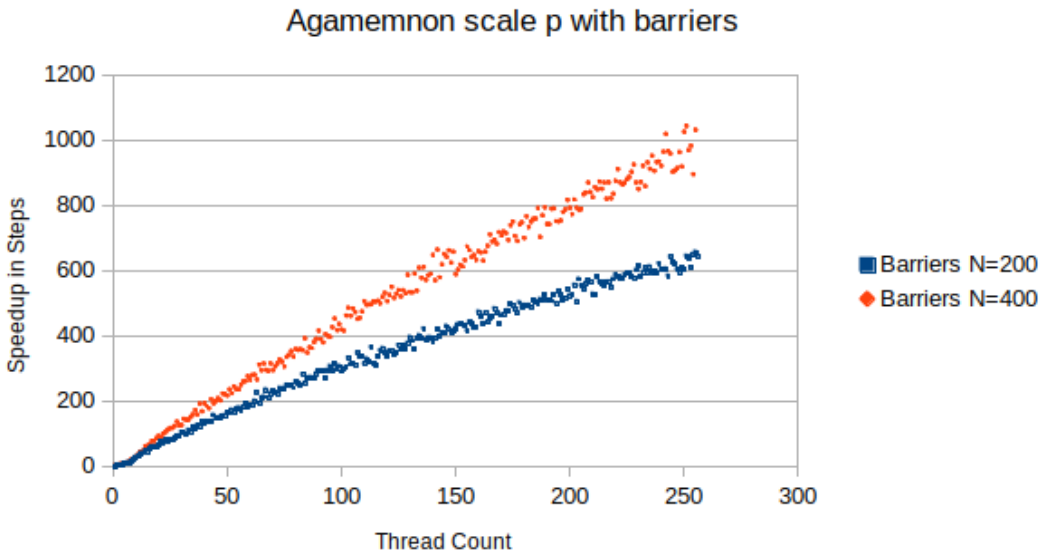

Figure 6.6: Comparison of speedup for various thread counts with a barrier implementation on 200 and 400 queens. Higher is better

6.4 Thread-Capped Lock-Based Implementation

Another idea to potentially improve the collaborative implementation is to increase the local load in each thread. We do this by using a comparatively low amount of instances, but assigning more than one solution to each thread. At every step, each local solution within every thread gets incremented sequentially and copy or reset operations only occur after a step through all local solutions. At the end of a step, each Instance attempts to activate all local solutions that have failed in this step by copying from other solutions within that same Instance. If no local solution can be copied from, it attempts to copy a solution from another Instance. Only when there is no other Instance with an active solution, all local solutions are reset.

We observe that a reset almost never occurs in this version. When copying from another instance, we need to make sure the solution is not changed while copying. We do this with a *copy_lock* like in the original implementation. It is possible that a lock on that Instance has been set in which case the operating system switches away from this thread while waiting for the lock to be reset. However, local copies are never blocked by a set lock because local solutions can only be changed by the same Instance they belong to. It follows that there is no need to switch active threads unless we need to copy from another Instance that currently has its lock set.
We hope that we can achieve a comparatively high solution count for using the inherent speedup of the collaborative approach while reducing the cost of frequent thread-switching. If we have x local solutions and y threads, we have a total of $x * y$ solutions. However, we also have a decreased speedup due to the fact that solutions are copied from other local solutions unless all local solutions failed in the last step. Because there are fewer local solutions, the same solution is copied more frequently than in the previous implementations. This leads to more solutions making identical random choices, which decreases the total probability of finding a correct solution. We hope that for some configurations of thread count and local solution count, the advantage of decreased system cost outweighs the reduced probabilistic speedup.

Fig. 6.7 shows a code sample of the central loop that is executed by every thread. It is similar to previous implementations in structure. The main additions are that every Instance now holds an array of solutions,

rather than just a single solution, and a counter *active_count* of how many local solutions are currently active. The counter is increased every time a solution is reset, or copied to and decreased when it fails to extend. Rather than only extending its one solution, each thread now calls an increment operation for all its local solutions and terminates if any of them reach the N-th step. After a step through every local solution, we check if any local solution has failed to extend in this step.
If *active_count* is greater than 0, meaning there are still active local solutions, a random active local solution is copied from for every inactive local solution in the function *Instance::copy_solutions_within()*. At the end of this function all local solutions are active again.
Likewise, if there are no active local solutions, the function *Instance::copy_solutions_outside()* goes through all failed local solutions and selects a random other Instance and copy a random active solution from that Instance. The implementations for both local copy functions are similar to Fig. 5.6 and are not shown in detail. Of note is only that *copy_solutions_outside()* locks the *copy_lock* of the Instance it attempts to copy from and *copy_solutions_within()* locks the Instance's own *copy_lock* because it changes the values of local solutions, like *increment* and *reset*. This way, no other Instance may copy partially copied solutions. In case no solution can be copied, all local solutions are reset.

```
while(!success){
  for(int sol_num = 0; sol_num < instance[id]->solution_count; ++sol_num){
    failAny = false;
    if(success){
      goto terminate;
    }

    if(!instance[id]->increment(sol_num)){
      failAny = true;
    }

    if(instance[id]->current_row(sol_num) == N){
      success = true;
    }
  }

  if(failAny){
    if(instance[id]->active_count > 0){
      instance[id]->copy_solutions_within();
    }
    else if(!instance[id]->copy_solutions_outside()){
     instance[id]->reset_all_solutions();
    }
  }
}
```

Figure 6.7: Central loop of thread-capped lock-based implementation

6.4.1 Testing Thread-Capped Implementation

The table in Fig. 6.8 shows test results for thread counts from 8 to 128 with a local solution count of 1 to 128. The table values are the average real-time in seconds required to solve a 500-Queens problem, sampled 500 times. For one thread, this implementation is almost identical to the initially presented lock-based implementation because it only copies from the single available solution in other instances, and never from its own solutions. Our initial hope was that a local solution count greater than 1 might reduce the required time.
However, this does not seem to be the case in almost all data points. In the 8, 16 and 128 thread versions, an increase of local load always leads to a longer execution time. This is probably caused in part by the reduced collaborative advantage due to local copying as explained above. Furthermore, if there are fewer threads to

switch to, the operating system's ability to mask delays of loading data from memory by switching threads is diminished. This effect is especially pronounced for low thread counts because the increase in required time is comparatively larger for 8 and 16 threads than 128 threads for up to 16 local solutions. With 128 threads, the operating system has significantly more threads to choose from and can mask long memory load times better. For 128 threads, the required time only increases significantly for 32+ local solutions . This is likely due to the small problem size of only 500. A single step through 32+ local solutions for all 128 threads takes up a significant amount of time before any copying is done. That way, most of the statistical advantage is lost.
Note that for 32 and 64 threads, the execution time for 1 local solution is significantly worse than for the other thread counts with 1 local solution. This is likely due to the lock-based nature of the approach. The previously observed problem of a break-down in speedup for certain thread counts is almost certainly to blame here because the time is reduced again for 128 threads. For both 32 and 64 threads, a local solution count of 2-16 leads to a faster execution. In fact, the execution time for 32 threads with 2 local solutions is barely higher than that of 16 threads with 2 local solutions. The break-down of speedup seems to vanish entirely for any local solution count greater than 1.
In conclusion, increasing local load seems to possess no advantage on this machine except in the special case of increasing the local solution count from 1 to 2 for problematic thread counts in the lock-based implementation. There is especially no advantage to locking the thread count to the hardware limit (8 in this case) and increasing the local load to prevent scheduling issues. A lock-based implementation with 2 local threads may be used over one with a single local solution. It performs slightly worse than a single local solution for most data points but doesn't seem to have the aforementioned problems of worse-performing thread counts. It still performs much better than our barrier-based implementation.

	Local Solutions							
Thread Count	1	2	4	8	16	32	64	128
8	0.131	0.196	0.342	0.385	0.421	0.495	0.567	0.584
16	0.039	0.063	0.082	0.109	0.155	0.219	0.281	0.436
32	0.302	0.07	0.083	0.112	0.175	0.252	0.444	0.769
64	0.367	0.12	0.148	0.182	0.262	0.385	0.675	1.195
128	0.2	0.213	0.229	0.259	0.378	0.584	1.053	2.03

Figure 6.8: Average seconds required to solve a 500 queen problem with various thread counts and local solution counts on an 8-core CPU

7 Experiments on a GPU

There is an inherent overhead to starting and managing CPU threads that counteracts some of the increased speedup in a real-time scenario. This makes it unlikely that starting very high thread counts, such as thousands of threads, is an improvement for solving problems quickly. Because of this, we implement and test both RIC approaches on a GPU using CUDA. A GPU has almost no marginal costs for starting additional threads once the initial startup cost for some threads is paid, and is designed to be used with many thousands of threads simultaneously. Our hope is to use the higher thread count that a GPU can offer to make use of the inherent convergence increase from running more threads concurrently. However, each GPU thread itself operates slower than any one CPU thread, and it has special limitations that make arbitrary computation challenging.

7.1 Special limitations of GPU Programs

GPUs have a number of advantages and limitations we need to consider if we want to write an efficient algorithm for a GPU. A general overview for writing efficient GPU programs is given in [Cor19a]. Notably, modern GPUs have a lot more computing units than modern CPUs. NVIDIAs Titan RTX card has a total of 4608 computing units, which they call CUDA Cores [Cor19b]. Each of these units is capable of executing some computation, but they are not comparable to a CPU core. They are not able to perform independent computation like CPU cores can, instead being bound into groups called *warps* that all have to execute the same instruction before being able to move to the next instruction. They may each execute that same instruction with different data though. This is called Single-Instruction-Multiple-Data (SIMD) processing. The amount of threads in a warp is hardware-specific, though usually 32 in NVIDIA GPUs. Because all threads within a warp need to execute the same instruction, branches in the code that cause threads within a warp to execute different instructions force some of those threads to wait until the execution of the branch is finished. This way, the concurrency of execution provided by the hardware is lost. If at all possible, parallel programs for GPUs must avoid having branches that affect a warp. Each core is also significantly slower than a CPU core, so writing programs for GPUs is only reasonable if it is possible to define tasks for a very high number of threads. Thankfully, our approach can be scaled infinitely, given we have enough memory.

GPUs have a limited amount of specific memory for each thread, and they rely on a large number of registers for most local computation. This is not enough memory to hold a solution, so we can only use this memory to store some variables. Threads on a GPU are also organized into blocks, which are executed within close proximity and most notably there is also a special on-chip memory that is accessible to all threads within a block, called shared memory. Shared memory is very small, but allows for fast access to data. On the machine we used, this can be set to be at most 48KB of data. To execute a block in a sensible manner, we would want to execute a multiple of the warp size, so at least 32 threads. That way, no warp is under-full and we fully use the hardware-given parallelism. If we were to save a solution as a simple array of `unsigned short`, which have a size of 2 Byte, we would need to allocate at least 64 Byte of shared memory per queen per block. Doing this, we could only allocate enough space for solving a problem with $48000/64 = 750$ queens. Problems of that size are solved almost instantly anyway, so there is not enough shared memory to store the solutions for interesting problem sizes. Thus, we opted not to use shared memory to store our solutions.

It follows that for storing solutions, we must rely on global memory, which is shared across all threads, but has significantly slower access times than shared memory because it is off-chip. To efficiently use global

memory, we have some hardware assistance in accessing data. As explained before, SIMD instructions are able to fetch or write a certain amount of continuous memory at once. GPUs offer these SIMD instructions for accessing data that a whole warp needs, and distributing it across the threads of that warp. On NVIDIA GPUs, this is typically 32 Bytes of continuous memory that is accessed in a single instruction.

Our first attempt to port the CPU implementation stores the solution for each thread in continuous memory, and different solutions are appended to one another. So if each queen takes 2 Bytes to fetch, we would only ever fetch one queen with an instruction that is able to fetch 32 Bytes of memory. If we wanted to access a single queen for each thread of a warp, we would need to use 32 instructions to fetch all those values. However, if we only ever start the kernel with a block-size that is a multiple of 32, we know that each warp tries to access 64 Bytes of memory when accessing a queen. So we can store the first queen for each thread in continuous memory, then following that store all second queens in continuous memory, etc. If we need 64 Bytes per warp in one expression to access a queen, that can be fetched or written in 2 instructions that access 32 Bytes each. This is called coalescing of global memory access, and should increase performance significantly.
Fig. 7.1 shows a graphical example with 3 queens each and 4 threads in a warp. Each block of colored memory is one queen that belongs to a certain thread. The different colors of memory signify to which thread that memory belongs, and each arrow is a memory access instruction. Curly brackets signify that the memory locations between them are accessed by a single instruction.

This global memory is separate from the main memory, and must be explicitly allocated by the CPU program. Data must be explicitly transferred, if transfer from main memory to the GPUs global memory is supposed to occur. This is an expensive operation, so there is a significant one-time cost at startup of any GPU program that needs to access data in its kernel. Allocations are performed on the CPU in a pre-processing step. GPU programs are executed in a *kernel*, a special function that is called from the CPU to execute on the GPU. For CUDA, the amount of threads to execute can be specified by how many blocks are to be started, and how many threads make up a block. The total thread count is the number of blocks * the number of threads per block.

Because of these limitations. GPUs are usually used to calculate specific sub tasks in massively parallel scale, rarely to compute whole algorithms. However, utilized correctly, GPUs can speed up computation significantly and allow for better scaling. We hope that our approaches can be scaled enough to gain a speedup for executing many instances at once, as compared to the slow-down of using slower cores. RIC itself can be executed many times entirely independently, which we believe might make it well suited for execution on a GPU.

7.2 GPU Implementations

7.3 Structure and Setup

As described before, we make some significant changes to the code in order to have a reasonable execution time.

- First, we significantly simplify the structure of the program. We were using a class Solution that holds member arrays for the queens and diagonals as well as member functions for increment and reset. However, this leads to all the data being in continuous memory for each thread. We need to interleave the structure for each of these arrays to allow coalesced memory access. So we avoid using the class Solution entirely. Instead we simply have two arrays that hold the diagonals and one array of queens for *all* the threads. These can be pre-allocated on the host, and initialized in a setup step within the GPU kernel. The arrays *dp* and *dn* need to hold 2*N-1 values for each thread, so the are bool arrays of length *thread_count*(2*N-1)*. The *queens* array is allocated as an array of `short` with size N for each thread.

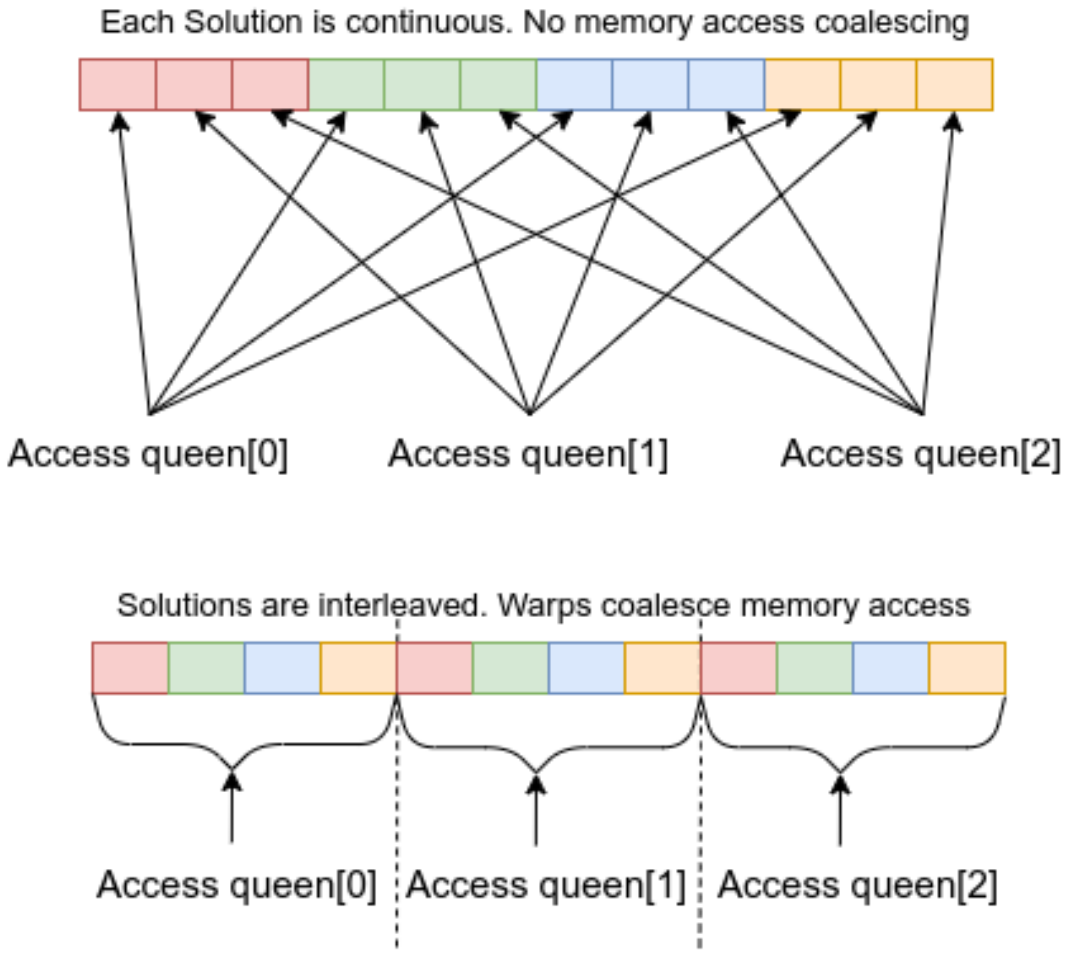

Figure 7.1: Coalescing global memory access by interleaving solutions. Example shows 4 threads in a warp on a problem size of 3

To access memory, we need to find the position of the element we are searching for. Each thread has a unique id, which is calculated as $int\ id = threadIdx.x + (blockIdx.x * blockDim.x)$ within the kernel. To access the queen on row x for a specific thread, we calculate the index as $queens[x * thread\_count + id]$, because there are always *thread_count* queens of the same row in continuous memory before the queens for the next row start. To access the diagonals for any one thread, we can calculate the diagonal number of that thread and queen position as $ip = i + queens[i * thread\_count + id]$ (for diagonals with positive slope) and then calculate where it is positioned in the dp array as $dp[ip * thread\_count + id]$. In addition, we also pre-allocate an array of random number engines in global memory, one for each thread.

- The second significant change is to the increment operation. Of course, we need to adjust how we access queens and diagonal array like described above. Furthermore our current increment operation that is similar to Algorithm 2 iterates through all queens in a solution and swaps them to the front if it is a possible extension. Each swap is a branch that causes execution in a warp to be sequential for some threads. If possible we would prefer for each thread within a warp to execute the same instruction as much as possible. The implementation shown in Fig. 7.2 uses a separate array of indices *swappable_offsets* for each thread which point to possible extensions, rather than swapping them to

the front. This array is gradually built up with all swappable offsets starting at the current row. At every step, we first do a provisional save of the queens offset to the current front of the indices array. *k* is the front of the offsets array. We increase the front of the indices array only if both diagonals of the position, were it to be chosen as the extension, are not currently occupied. If we do not increase the front, the provisional selection is overwritten in the next iteration. We start the indices array at 2 so we can do x % (k-1) later without doing x % 0. Once all iterations through the rows are finished, we need to choose one random extension out of all the possible swappable offsets. In particular, we need to consider that some threads may not have found any extensions, which is the case if k is 2. For any k > 2, we found at least one possible extension. The expression (rng.random() % (k - (k>2) - 1)) + (k>2) + 1 evaluates to (rng.random() % 1) + 1 = 1 for k = 2. At *swappable_indices[1]*, we simply save the number 0 in a setup step, so j becomes i + 0. In that case, we swap the queen with itself, meaning there is no change. If k > 2, the expression evaluates to (rng.random() % (k-2)) + 2, which has values in {2,...,k-1}. That way, the random choice is always one of the previously set possible extension. The rest of the operation is like before, except that we only return true if k > 2, meaning that we found a possible extension. In this implementation, all threads always execute the same instructions and never diverge.

- Because we are using this thread_count variable a lot, and it is constant, we can put it into a special cached section of global memory called *constant memory*. Values in constant memory are be broadcast to all threads within a warp in one single instruction when all threads access the same variable, and it is slightly faster than global memory.

- It may be possible to speed up resets of a solution if only some threads within a warp failed to extend. In that case, all non-failing threads within the warp have to wait while the other threads reset. Fundamentally, a reset operation consists of resetting the diagonal arrays. This is 4*N-2 memory write operations for resetting 4*N-2 bytes of memory (assuming *sizeof(bool)* is 1). With 1 resetting thread and 31 waiting threads, we would be able to cut this down to $\lceil \frac{4*N-2}{32} \rceil$ instructions due to memory access coalescing. However, this would require the diagonal arrays to be in continuous memory for each thread so it can be fetched in SIMD operations. Furthermore, we would need some additional resources to keep track of which threads have to be reset within each warp and which order to reset if more than one thread did fail. This could be done efficiently by using shared memory, but requires extra instructions for a small advantage. Furthermore, the more threads within a warp need to be reset, the lower the advantage becomes. If more than one thread resets with an interleaved memory structure, the reset can be coalesced even in that. In case more than half of a warps threads need to be reset, it would be advantageous again to have the diagonal arrays stored in an interleaved fashion. Because of these reasons, we decided not to split up resets between threads in a warp. This lends a minor advantage to the naive version, because each thread is more likely to reset in that version, making it more likely that the access pattern benefits a fast reset.

```
__device__ __inline__ bool increment(RNG& rng, int id, short& i){
  int k = 2;

  for(unsigned short j = i; j < N; ++j){
    int jp = i + queens[j*thread_count+id];
    int jn = i - queens[j*thread_count+id] + N-1;

    //Provisionally save this queen, gets overwritten if k isn't increased.
    swappable_offsets[k] = j-i;

    //If no diagonal is threatened, increase k. Converts true to 1.
    k += (!dp[jp*thread_count+id] && !dn[jn*thread_count+id]);
  }

  int random_extension = (rng.random() % (k - (k>2) - 1)) + (k>2) + 1;
  int offset = swappable_offsets[random_extension];
  int j = offset + i;

  unsigned short tmp = queens[i*thread_count+id];
  queens[i*thread_count+id] = queens[j*thread_count+id];
  queens[j*thread_count+id] = tmp;

  int ip = i + queens[i*thread_count+id];
  int in = i - queens[i*thread_count+id] - N-1;
  dp[ip*thread_count+id] = true;
  dn[in*thread_count+id] = true;

  i += (k>2);

  return (k>2);
}
```

Figure 7.2: GPU increment implementation without divergent paths

7.3.1 Naive GPU RIC

The actual naive RIC kernel is similar to the previously presented parallel block in Fig. 5.2, only changed slightly in setup and memory access as described before. It is not be presented here. Originally, we thought that the naive RIC approach might be better suited for GPU execution than the collaborative approach, because there is almost no need to wait for other threads and no synchronization. Each thread operates completely independently, and the only divergent behavior is for a reset. However, we failed to consider that the naive RIC version naturally diverges in the current progress of the solution in each thread. All threads within a warp stay in lockstep until one of them resets. The reset itself is fast enough to not meaningfully diverge the execution, but after the reset, that thread starts from an empty sequence and the first row. Due to this, it no longer accesses queens on the same row in increment operations as the other threads in the same warp. This breaks the coalesced memory access described above. Each thread that is thus diverged brings an additional memory access instruction for every access. Some divergent behavior may be sped up by using shared memory instead for those divergent warps, but almost all threads within a warp will eventually diverge in their row, simply because the problem is big. We see no solution to this fundamental problem, which would make memory access eventually regress close to what it was in the simple implementation. Solutions are still found considerably quicker than if we had continuous memory, but slower than the collaborative approach.

7.3.2 Collaborative GPU RIC

Fundamentally, the collaborative lock-based approach used for the CPU implementation can not be ported efficiently to a GPU. CUDA does not provide any lock implementations, but even if they did it would not

work well. Because of the coupled nature of a warp, when trying to lock a thread within a warp, all other threads in that warp would also be blocked. Because each thread has to lock on every increment and reset operation, this would lead to constant locking of every thread and very slow execution. So we use a barrier implementation that is is similar to the CPU barrier implementation. The main advantage of this is that we know that at the start of every step, all threads are at the same current row because they either incremented their own solution, copied a solution that incremented or reset along with all other threads. This way, there is full memory access coalescing in every increment operation. The actual kernel also closely follows from the previously presented CPU implementation, with *failAll* being a variable in shared memory.

The forced synchronization makes this approach significantly slower for equal thread count as compared to the lock-using version. However, the inherently higher possible thread count of a GPU and better convergence for higher thread count counteracts that slow-down to some degree. However, there are other inherent limitations that limit the scalability this implementation. Most importantly, barriers in CUDA are limited to block-level synchronization. Build-in synchronization across blocks is possible in newer CUDA versions, but we have no access to any device that can use these features. We have to assume that different blocks operate completely independently, and can not be meaningfully synchronized in an efficient manner. However, there is a hardware limit for how many threads can be executed in one block. On our machine, this is 1024. This puts a hard cap on the number of threads that can be executed collaboratively with this approach. This is problematic, because each GPU thread is already slower than a CPU thread. Starting fewer threads in a single block and starting more than one block is also generally preferable where possible, which also soft-caps the amount of threads we should start. These considerations limit the usefulness of our GPU implementation significantly.
To combat the hard cap of threads, 1024 threads can be run cooperatively in each block, while running as many blocks as possible independently. This requires communication between blocks for terminating early in case of success, which is possible by global memory. Each thread only copies solutions from within their block. However, this has no additional cooperative speedup, only a speedup from faster convergence due to an increased randomization. We found that starting more than a single block for the collaborative approach did not increase real execution time any further than starting the maximum amount of threads in a single block.

7.3.3 The GPU test

Because we already tested the inherent speedup for step counts, we only want to know, if collaborative execution is still preferable to naive execution on a GPU and whether a GPU version can solve reasonably large problems in comparable real-time to a CPU version. However, finding sensible problem sizes and thread counts for testing the CUDA implementation is difficult, because running it is only interesting for high thread counts. It has no advantage for low thread counts because each thread is slower than a CPU thread. Furthermore, GPU programs have a significant overhead for transferring data, starting and stopping a kernel. For low N, this overhead overshadows any actual computation. So results for low N do not accurately represent the actual computation but are influenced mainly by outside factors. Finally, testing for high N with enough samples to have a statistically significant result takes a long time due to the exponential nature of finding solutions. This leads to us testing only comparatively few data points.

Fig. 7.3 shows the results of experiments conducted on an NVIDIA Tesla K20c graphics card, with a CPU test of the barriers implementation on the same 8-core CPU as before for comparison. Every data point is sampled 500 times for problem sizes ranging from 400 to 470 queens. The y-axis shows the real average time required in seconds. There are 6 GPU data series, for the naive and collaborative version for 256, 512 and 1024 threads, respectively.
We observe that the naive 256 threaded version is the worst performing version by a large margin. The 1024 threaded naive version performs slightly better than the 512 threaded naive version. All collaborative versions are about twice as fast as any naive version, and comparatively close together which is unexpected. There seems to be less of a collaborative advantage for high thread counts for GPU execution. This might be due to the fact that starting more than 512 threads in a single block is generally not a good idea in CUDA.

More blocks with fewer threads each tend to perform better, but we can not synchronize across blocks, so we may not use this advantage. Accordingly, the collaborative version with 512 threads performs better than the 1024 threaded version. As expected before, the general advantages of collaborative execution as well as the better memory access coalescing lead to collaborative execution being preferable to naive RIC even on our GPU.

However, the lowest data series corresponds to the 64-threaded barrier collaborative CPU implementation. It performs slightly less than twice as well and the barriers CPU implementation already performs worse than the lock-based CPU version. This does not necessarily mean that a GPU implementation must always perform worse than a CPU implementation though. Hardware architectures for GPUs and CPUs are very different, and comparing the capabilities of the specific CPU and GPU is hard, so one might be significantly better than the other which would influence these results. The fact that our GPU implementation is able to perform almost as well as our CPU implementation is great. In more modern GPUs that allow for cross-block synchronization, this allows us to scale the collaborative approach to significantly more threads than any CPU would be able to use effectively. Depending on the specific problem and hardware available, it may thus be preferable to use a GPU over a CPU.

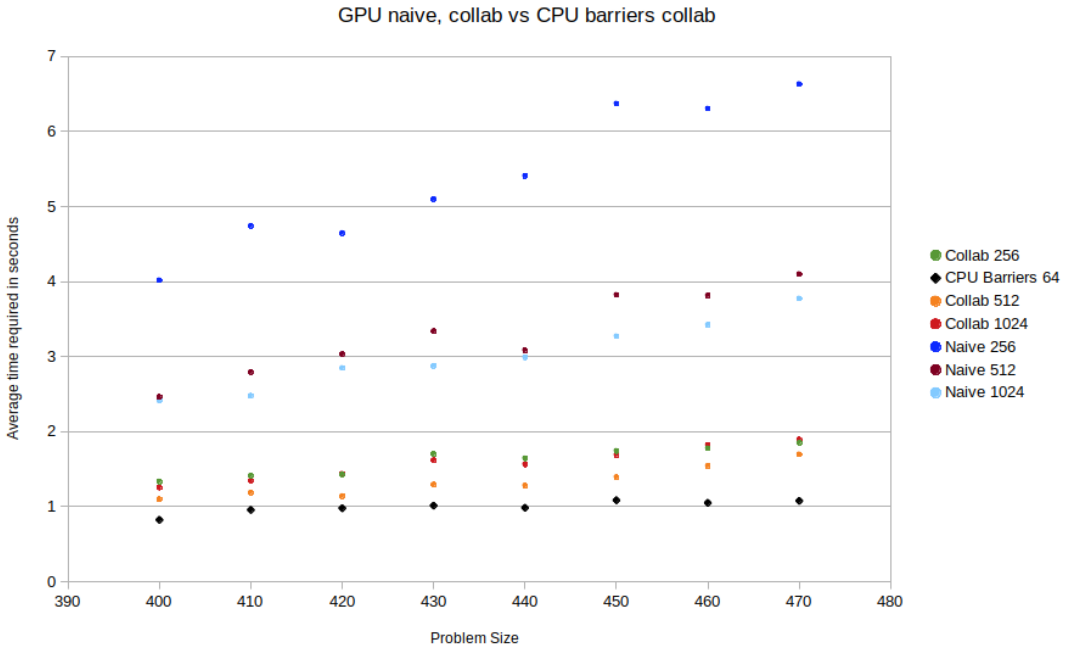

Figure 7.3: GPU real-time comparison between various naive and collaborative thread counts with 64-threaded CPU barrier implementation for comparison

8 Conclusion

This thesis presents and evaluates a collaborative scheme to improve parallel execution of Randomized Incremental Constructions for solving Constraint Satisfaction Problems, a vast class of problems that find many real-world applications. The general structure of RIC and the collaborative execution is simple and should be easy to implement for any given CSP. Furthermore, its structure is perfectly fit for highly parallel execution, which is increasingly important due to trends of increased parallel capability in modern hardware. Evaluation is done through a series of randomized tests, that sample each data point of various problem sizes and thread counts as the average of several hundred repetitions of any given execution. This way, we can make conclusive statements despite the randomized nature of the presented algorithms.
Our experiments show that collaborative execution is preferable to naive execution in all tested circumstances. In particular, the collaborative approach scales far less quickly with the problem size, which counteracts the exponential nature of RIC. For solving large problems, the difference only grows wider, allowing us to solve much larger problems in a reasonable time-span. In order to fully use this statistical advantage, we think a quick implementation using locks is possible but we have not been able to present one without issues here. The implementation we showcase has a reduced speedup for certain thread counts, but an increased speedup for values beyond that break-down. This leads us to believe that the issue is specific to our implementation, the used framework or hardware, and not a general problem with lock-based implementations. A much slower barrier-based version is still preferable to naive RIC execution and lets us conclude that collaborative execution has increased speedup values even far beyond hardware capabilities. A lock-based version with increased local load may be preferable to a barrier implementation, but generally performs worse than a lock-based implementation. We also aimed to test whether collaborative execution is sensible on a modern GPU architecture, although it has several relevant limitations that make the execution slower. We can conclude that on a GPU, collaborative execution is still preferable to naive execution but most likely not preferable to a CPU version. However, the results are surprisingly close so a GPU implementation may yield comparable or better results than a CPU implementation given the right hardware and proper optimization. This leads us to believe that collaborative RIC is especially useful on modern or future hardware with even larger parallel potential.
The results are conducted on a specific CSP, and whether they are applicable to other CSPs is yet unclear. We believe that collaborative execution should still be preferable for other CSPs as we used no characteristic that is specific to the N-Queens problem for the collaborative implementations that we did not also use for the naive implementations. Furthermore, the tests were originally planned to be conducted on two machines, but due to the current Covid-19 crisis we had no access to the better machine after some initial tests. The results presented might be dependent on the specific hardware used, and especially the latter GPU tests and CPU to GPU comparison might not be identical on other machines. However, we tested the step count required to solve a problem, which should be almost entirely hardware-independent. We saw significant speedups in the step-count for higher thread counts, which leads us to believe that our results are representative regardless of hardware specifics, at least for common modern CPU and GPU architectures.
In the future, it would be interesting to test our results against different CSPs to see if the collaborative advantage is similar or perhaps even greater for other problems. It would also be interesting to see if the collaborative approach may be implemented for distributed systems. Distributed systems allow for much greater scale of computation, but have comparatively high communications cost. This trade-of would certainly be interesting to explore.

Declaration of Academic Integrity

I hereby confirm that this thesis on

_____

_____ is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

_____

(date and signature of student)

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

_____

(date and signature of student)

Bibliography

[Ahr01] W. Ahrens. *Mathematische Unterhaltungen und Spiele*. B.G. Teubner, 1901, pp. 114–156. URL: http://www.archive.org/details/mathunterhaltung00ahrerich (cit. on p. 2).

[Alw17] Roshan Alwis. *Parallel Matrix Multiplication [C][Parallel Processing]*. [Online; accessed 16-March-2020]. 2017. URL: https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27 (cit. on p. 12).

[AM15] Andrei Krokhin3 Andrei A. Bulatov1 Venkatesan Guruswami2 and Dániel Marx4. *The Constraint Satisfaction Problem: Complexity and Approximability*. [Online; accessed 2-February-2020]. 2015. URL: https://drops.dagstuhl.de/opus/volltexte/2016/5671/pdf/dagrep_v005_i007_p022_s15301.pdf (cit. on p. 2).

[Ban16] Jannes Bantje. *Vorlage für eine Bachelorarbeit an der WWU Münster*. [Online; accessed 4-February-2020]. 2016. URL: https://github.com/JaMeZ-B/template_bachelor_thesis_wwu (cit. on p. 1).

[Bla] Paul E. Black. *Fisher-Yates shuffle*. [Online; accessed 1-March-2020]. URL: https://xlinux.nist.gov/dads/HTML/fisherYatesShuffle.html (cit. on p. 21).

[Boa] OpenMP Architecture Review Board. *The OpenMp API specification for parallel programming*. [Online; accessed 4-February-2020]. URL: https://www.openmp.org/ (cit. on p. 17).

[BS09] Jordan Bell and Brett Stevens. 'A survey of known results and research areas for n-queens'. In: *Discrete Mathematics* 309.1 (2009), p. 2. ISSN: 0012-365X. URL: http://www.sciencedirect.com/science/article/pii/S0012365X07010394 (cit. on pp. 2, 4).

[Byr14] Michael Byrne. *Memory Is Holding Up the Moore's Law Progression of Processing Power*. [Online; accessed 16-March-2020]. 2014. URL: https://www.vice.com/en_us/article/3dkkx8/memory-is-holding-up-the-moores-law-progression-of-processing-power (cit. on p. 13).

[Cor19a] Nvidia Corporation. *CUDA C++ Best Practices Guide*. [Online; accessed 17-March-2020]. 2019. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#preface (cit. on p. 33).

[Cor19b] Nvidia Corporation. *NVIDIA TITAN RTX*. [Online; accessed 16-March-2020]. 2019. URL: https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/ (cit. on p. 33).

[Dev19] Advanced Micro Devices(AMD). *AMD EPYC™ 7002 Series Processors*. [Online; accessed 16-March-2020]. 2019. URL: https://www.amd.com/en/processors/epyc-7002-series (cit. on p. 13).

[DP87] Rina Dechter and Judea Pearl. 'Network-based heuristics for constraint-satisfaction problems'. In: *Artificial Intelligence* 34.1 (1987), pp. 2–8. ISSN: 0004-3702 (cit. on pp. 2, 4).

[Erl] Friedrich-Alexander Universität Erlangen-Nürnberg. *Moore's Law*. [Online; accessed 4-February-2020]. URL: https://moodle.rrze.uni-erlangen.de/pluginfile.php/14438/mod_resource/content/2/07_SIMD.pdf (cit. on p. 13).

[HE80] Robert M. Haralick and Gordon L. Elliott. 'Increasing tree search efficiency for constraint satisfaction problems'. In: *Artificial Intelligence* 14.3 (1980), pp. 263–313. ISSN: 0004-3702. URL: http://www.sciencedirect.com/science/article/pii/000437028090051X (cit. on p. 5).

[HTA92] A. Homaifar, J. Turner and S. Ali. 'The N-queens problem and genetic algorithms'. In: *Proceedings IEEE Southeastcon '92*. 1992, 262–267 vol.1 (cit. on p. 3).

[Kem18] Anil Kemisetti. *Solving Sudoku … Think Constraint Satisfaction Problem.* [Online; accessed 14-April-2020]. 2018. URL: https://medium.com/my-udacity-ai-nanodegree-notes/solving-sudoku-think-constraint-satisfaction-problem-75763f0742c9 (cit. on p. 2).

[LS92] Donald E. Knuth Leonidas J. Guibas and Micha Sharir. 'Randomized Incremental Construction of Delaunay and Voronoi Diagrams'. In: *Algorithmica* 7 (1992), pp. 381–413 (cit. on p. 8).

[Per19] Tekla S. Perry. *Forget Moore's Law—Chipmakers Are More Worried About Heat and Power Issues.* [Online; accessed 13-March-2020]. 2019. URL: https://spectrum.ieee.org/view-from-the-valley/semiconductors/design/power-problems-might-drive-chip-specialization (cit. on p. 12).

[Rat12] Justin Rattner. *Tera-scale Computing - A Parallel Path to the Future.* [Online; accessed 16-March-2020]. 2012. URL: https://software.intel.com/en-us/articles/tera-scale-computing-a-parallel-path-to-the-future/ (cit. on p. 13).

[Saf20] Matt Safford. *How to Buy the Right CPU: A Guide for 2020.* [Online; accessed 16-March-2020]. 2020. URL: https://www.tomshardware.com/reviews/cpu-buying-guide,5643.html (cit. on p. 12).

[Sei91] Raimund Seidel. 'Small-dimensional linear programming and convex hulls made easy'. In: *Discrete and Computational Geometry* 6 (1991), pp. 423–434. URL: https://link.springer.com/article/10.1007/BF02574699 (cit. on p. 8).

[SG90] Rok Sosic and Jun Gu. 'A Polynomial Time Algorithm for the N-Queens Problem'. In: *SIGART Bull.* (Oct. 1990), pp. 3–4. DOI: 10.1145/101340.101343. URL: https://doi.org/10.1145/101340.101343 (cit. on p. 4).

[SG91] Rok Sosic and Jun Gu. '3,000,000 Queens in Less than One Minute'. In: *SIGART Bull.* 2.2 (1991), pp. 22–24. ISSN: 0163-5719 (cit. on p. 3).

[Slo] N. J. A. Sloane. *Number of ways of placing n nonattacking queens on n X n board (symmetric solutions count only once).* [Online; accessed 17-February-2020]. URL: https://oeis.org/A002562 (cit. on p. 3).

[Tar19] Carle Tardi. *Moore's Law.* [Online; accessed 4-February-2020]. 2019. URL: https://www.investopedia.com/terms/m/mooreslaw.asp (cit. on p. 12).

[WM96] Wm Wulf and Sally McKee. 'Hitting the Memory Wall: Implications of the Obvious'. In: *Computer Architecture News* 23 (Jan. 1996) (cit. on p. 12).