Parallel Execution of Backtrack Search with optimizations

Bachelor Thesis by Christian Kaysø-Rørdam s082918

Project number: 15

Supervisor: Michael Reichhardt Hansen

Table of Contents

Abstract

This project covers the implementation of a parallelized version of the generic backtrack search algorithm and it is used to find all or just a single solution to the N-Queen problem. Furthermore, various new heuristics and optimizations are explored. All implementations are tested and the results are posted and discussed. The project also explores how the .NET Framework may aid in the design on parallelized programs.

1 Introduction

1.1 Parallel execution and optimizations of BTS

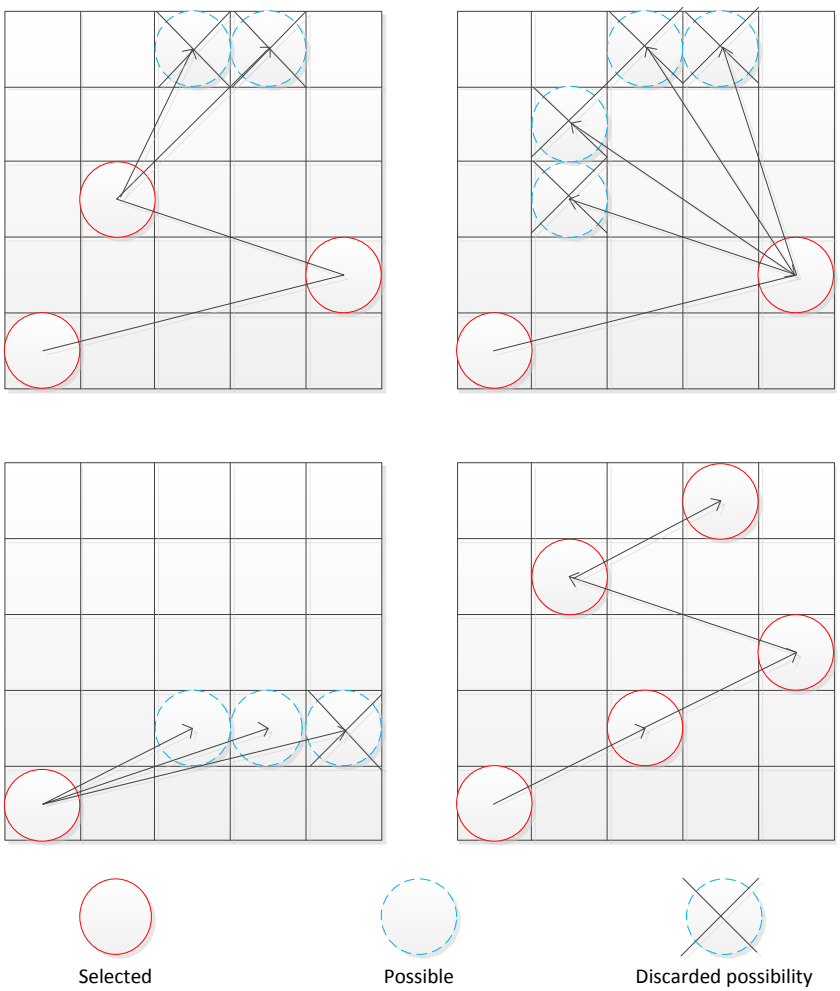
Most new computers come equipped with a multicore processor, which software developers can take advantage of to get better performance for their software. Even though multicore platforms are widely available and used in most places these days, taking advantage of the additional cores is still challenging. Using the additional cores may add a lot of complexity in the design process to an otherwise simple problem.

In this project, a parallelization of a generic backtrack search algorithm will be investigated and implemented and the algorithm will be used to solve the n-queen problems. A backtrack search algorithm has several other uses such as Sudoku and SAT solving, and a parallelization of the algorithm could easily be translated to be used with any of those. These problems are considered hard problems to solve, as they do not run in polynomial time. In the course of the project, various optimizations specifically for the n-queen problem will also be investigated and potentially implemented.

For the implementation of the algorithm and optimizations, the .NET platform will be used as the .NET Platform has many parallelization primitives which may aid in the development of a parallelized backtrack search and it may be interesting to see to what extent these can help in the development.

1.2 Example

Here is an example of using a backtrack search for finding a solution to the 5-queen problem.



In the first frame, we see a selection of three positions for queen placement, and the two possible selections at the top are both discarded because neither allow for a queen to be placed on the second row from the top. A backtrack is then performed, undoing the selection at the third row, and we see that there are no other possible selections. We backtrack yet again in frame three; discarding the selection at row four, note that not all possible selections are shown. In frame four, we simply select the only possible positions giving us a solution.

1.3 Background

Experiments have been successfully conducted to find optimizations that allow solving of large N-Queen problems in a realistic timeframe. Some of these optimizations include Simulated Annealing and Genetic Algorithms, which are general approaches for creating heuristics for specific problems and optimizing parameters. See Comparison of Heuristic Algorithms for the N-Queen Problem[1] by Ivica Martinjak and Martin Golub.

2 Principles of parallelization

2.1 Amdahl's Law

Amdahl's law[2] can be used to predict the speedup gained by parallelizing a program. The speedup describes the, hopefully, decrease in time taken to finish executing some algorithm by using additional cores. Given the relative size of the parallelized portion of the program P and the number of cores this is executed on N, the maximum speedup M is given as:

$$\frac{1}{(1-P)+\frac{P}{N}} = M$$

Where P is a number between 0 and 1 describing the parallel portion and 1-P gives us the serial portion. The serial portion of a program can be used to determine an upper limit for the speedup from parallelization, as the serial portion will never execute faster regardless of how many cores aid in the execution of the parallel portion. This means that even if the parallelized portion runs infinitely fast, due to infinitely many cores, the total execution time will be no faster than the execution time of the serial portion. The graph shows the speedup a program with differently sized parallel portions (0.95, 0.90, 0.75 and 0.50) can achieve when a number of extra cores are used in the execution.

[1] http://www.zemris.fer.hr/~golub/clanci/iti2007.pdf
[2] http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

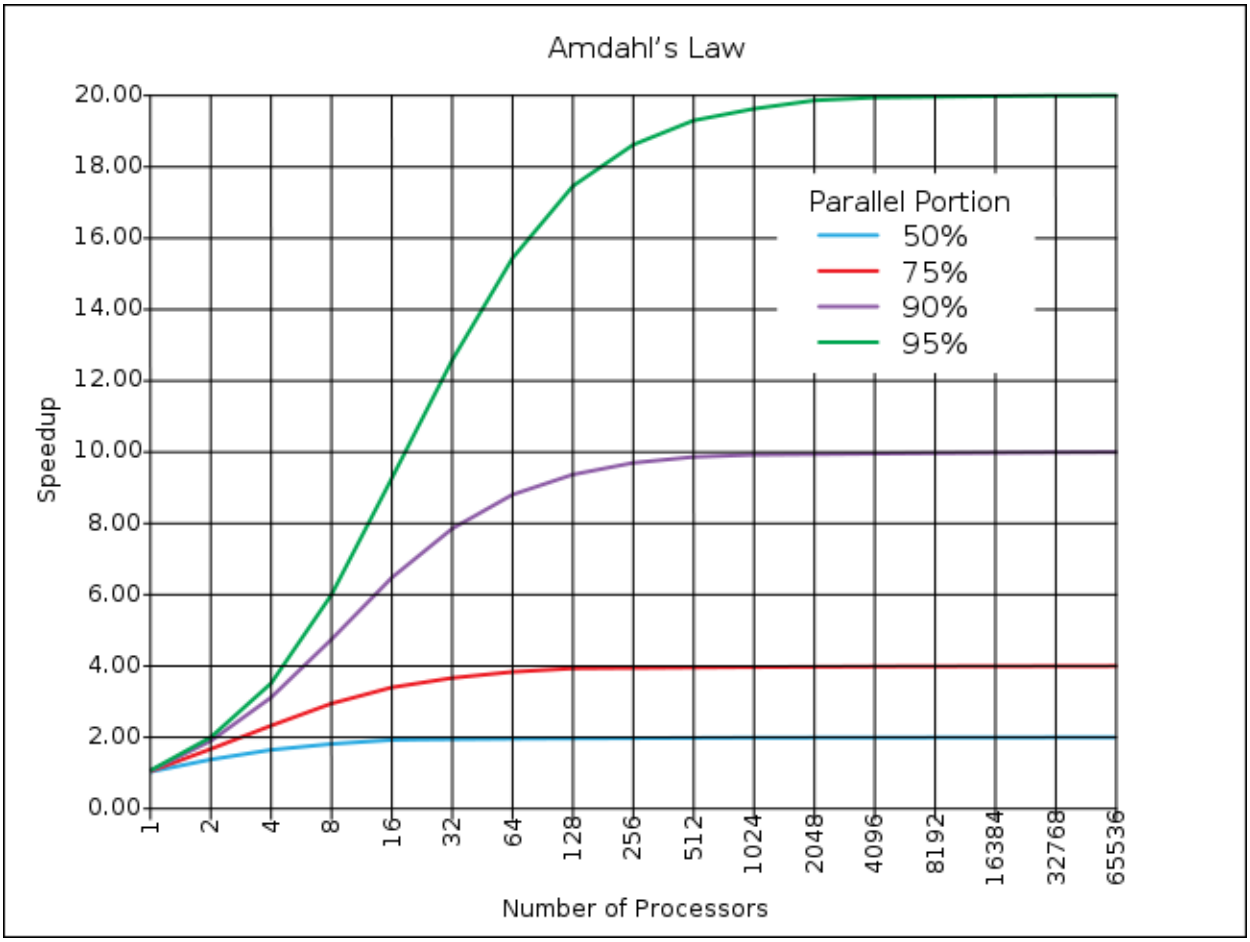


Figure 1, showing maximum speedup with differently sized parallel portion and various numbers of cores.[3]

The graph clearly shows that some problems are far less suited for parallelization than others, as even a program where half of it can be parallelized cannot achieve more than a two times speedup, regardless of cores used.

2.2 Delightfully parallel problems

A large subclass of parallelization problems can be described as being delightfully parallel[4]. A delightfully parallel problem is one where the individual tasks, which together make up the full solution, are fully or nearly fully independent of each other, and as such they do not share variables or rely on the result of the other tasks. This means that each part of the problem can be

[3] http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg
[4] http://msdn.microsoft.com/en-us/library/dd460693.aspx

carried out almost fully asynchronously, and therefore also being capable of utilizing the full potential of a multicore processor, given a problem that is large enough.

An example of a delightfully parallel problem could be to update each element in an array with a new value.

Due to the simplicity of a delightful problem, it is often worth try to a given problem in order to achieve a delightful problem. A common place to find delightful problems is in the area of tree-traversal, since trees are defined as having no cycles. One could imagine a binary tree where we know that the left and right sub-tree does not share any nodes, this lets us traverse both the left and the right side fully asynchronously and we would even be able to alter nodes in both sides without worrying about race-conditions or telling the other thread what was changed. This can be done because both sides of the binary tree are fully independent (if left and right was connected, there would be a cycle through the root). In fact, one would be able to traverse and make changes to any tree asynchronously by splitting it up into smaller sub-trees, as long as you make sure that no sub-tree is a sub-tree of any of the other sub-trees you made.

Many problems where one searches for a solution, can be defined as a finding a path in a tree to a node which fulfills a certain criteria, by traversing the tree and can therefore be done asynchronously. When looking for a solution in this way, at some point one of the threads will contain the full path needed to reach the desired node, not just part of the path. This happens because almost no data is shared between the various threads, and this differs greatly from the other type of parallel problems, the hard ones.

2.3 Hard parallel problems

Most of the threads used to solve a problem hard parallel problem will more often than not be dependent on the result of some other thread. This will rely heavily upon correctly synchronizing the various threads and gathering up and distributing the results found. The threads are very likely to share significant variables with each other, greatly increasing the needed synchronization. Not only are these kinds of problems very hard to provide a working implementation for, but it is also

very hard if not impossible to achieve the full workload on all available cores for the full duration of the time it takes to solve the problem. This is simply because of the synchronization needed; threads will frequently be in a locked state, where they will be waiting for some other thread to finish before they can proceed with their own work again. It is also very likely that some of the work that needs to be carried out, is simply too complex to parallelize and will therefore only make use of a single thread, leaving most the CPU idle for some time.

Many of these problems share a similar way of achieving a result; they task each thread with finding a partial solution or generating data which can be used to either find or verify partial solutions. The problem is that most of the threads will never be able to see the full picture, so they are capable of using the information they generate for anything on their own, a sort of control structure must be introduced to gather up information. The simple approach to making such a control structure is to use a thread purely for the purpose of keeping all other threads synchronized and collecting their data once they finish. This closely follows the master/slave pattern, frequently used in distributed computing. In fact, many of these problems are a kind of distributed computing, even if they only run on a single machine, the main problem is still distributed over many cores each providing its own small part of the final solution.

2.4 Basic CPU Architecture

Most desktop and server CPUs made today have more than one physical core. Each core may execute one thread at a time, possibly switching between threads to give the illusion that many applications may run at the same time. Many high end CPUs from Intel are equipped with Hyper threading technology, which allows the CPU the split a physical core into two logical cores, which may each execute a thread at the same time. The performance benefit gained from this Hyper threading technology may be modest in many applications[5].

[5] See conclusion in
http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

2.5 Threads

A thread is what executes a series commands specified in some programming language; these commands may also specify the creation of a new thread. Threads are contained inside a process and a process is associated with each application running on the machine, each process may have many threads. They are the very base of what allows us to run multiple applications on a single machine, where all applications appear to be running at the same time. This appearance of all applications running simultaneously is provided by the operating system in form of the scheduler. The scheduler is tasked with managing all the threads currently running on the computer, in such a way that they all get their share of time on the CPU. This scheduling of threads is needed because a single logical core is only capable of running a single thread at any given time; the thread the CPU is currently executing has much of the crucial data stored in the cache located on the CPU. When one thread has used up the time allocated to it by the scheduler, a new one will take its place on the CPU and this happens through a context switch, where the data from the old thread is extracted from the CPU and written in the memory and the data from the new thread is retrieved from the memory and stored in the cache. These context switches happen very often, but they do come at a cost, since moving data from one piece of memory to another takes up time, this overhead is of course minimized by the scheduler in such a way that threads that require a lot of work will get more time to do so, and threads that have nothing to do at the moment will get no time on the CPU.

Each thread must be individually created, and this process is very time consuming and so is tearing down a thread once it has finished executing. This is because a thread must be allocated a part of the memory upon creation, various handles and other required data must be generated for the thread and it must be added to the scheduler in order to get a chance to get executed. All of this will then have to be removed once the thread has finished, taking up even more precious time. There are smart ways of circumventing many of these costs, to make the decision of whether to use a thread or not becomes less of a concern for the programmer.

2.6 Complications with Parallelization

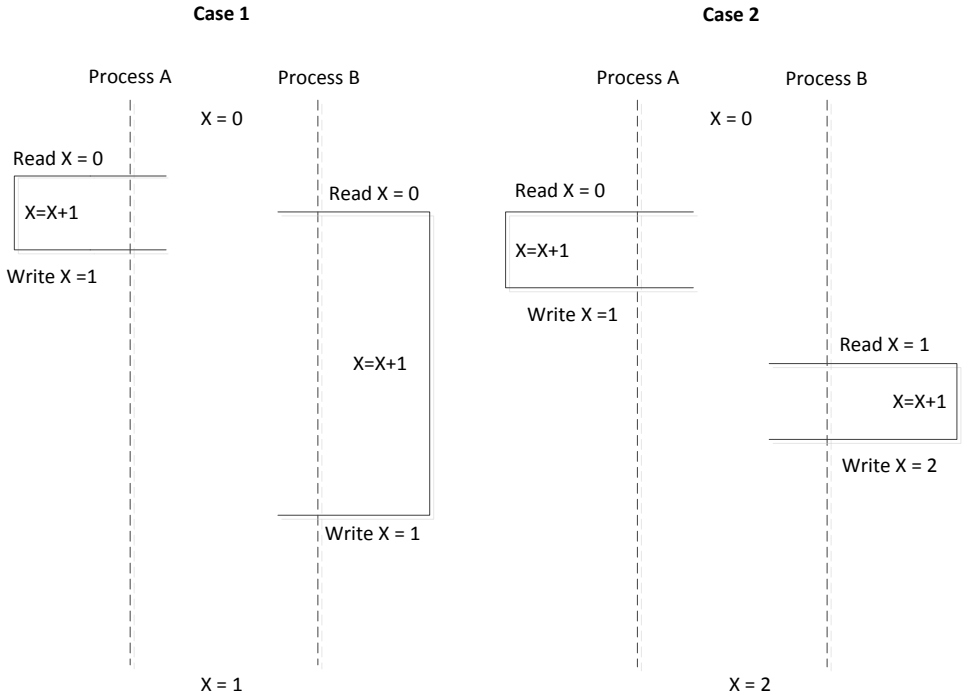
2.6.1 Shared data access

One of the most frequently occurring issues when constructing parallel programs is how to deal with shared data access, as improper parallel access to shared data can lead to race conditions. Race conditions arise from the interleaving of the side effects produced by the threads being executed. This interleaving of the side effects, are a direct result of the scheduling performed by the operating system and to some extent a lack of knowledge of the precise time each line of code actually takes to execute. In most cases, the scheduler only promises some degree of fairness, which means that all thread will be executed at some point, but it makes no promises in exactly which order or that it will not be temporarily switched out in favor of another process. All of this makes the interleaving unpredictable to the extent that safety measures must be worked in to the program, in order to ensure that any interleaving of the side effects will not cause the program to fail or produce incorrect results.

The most common place for race conditions is multiple threads writing to the same part of the same data structure at the same time, this could simply be writing some text to a file. We must guarantee that each thread will have exclusive access to the data or file it wishes to alter, while it alters it. All other threads would then have to wait for that thread to finish altering the data before they can alter it. The simplest way of providing this guarantee of exclusivity is in the form of a lock, which is written in the code around the various points where data shared by multiple threads is accessed. When a thread reaches a lock in the code, it will try to acquire it, and if no other thread is currently in possession of the lock it will be able to, otherwise it will have to wait for the lock to be released.

Race conditions can also be dealt with in other ways. One of the more obvious ones is to try and reduce the amount of data shared between threads, which not only reduces the number of race conditions but might also greatly reduce the complexity of writing the program. This is also why we chose to call some class of parallel problems delightful, because they have little to no need to share data between the threads, making writing a program that solves such a problem, delightful.

Here we see two cases of two processes adding one to the shared value X. The dashed line represents a time line and the vertical lines represents some actions taking place at some specific time.



X starts out as zero in both cases, but becomes one in the first case and two in the other. In case 1, the problem is that the read of Process B happens after the read of Process A and before the write of Process A. When Process A writes the updated value of X, Process B does not update its value of X, since it already performed a read, leading to the first update of X by process A to be lost.

In case 2, we see an example where X becomes two as expected, because the neither process has any overlap with the other at any time.

2.6.2 Thread overhead

When presented with a problem that greatly lends itself to being parallelized, such as a delightfully parallel problem, one might simply make a new thread for each of the tasks the problem needs solved. However, due to the way threads on the operating system functions, many limitations

apply to the use of threads and each task should be looked at to see if it is indeed worth making a thread purely for the purpose of executing it.

The first problem with making a lot of threads for each task, is the context switching. Because only a single thread can be executed per logical core, making more threads than there are logical cores available can lead to unwanted overhead due to the increase in context switching. A general guideline would be to only make the same number of threads as there are logical cores present on the machine the program is running on. Making more threads than there are logical cores is called oversubscription, which will slow down the program due to the increase in threads the scheduler will start doing context switches with the threads the program itself is using, which just wastes time.

In some cases, a thread in a parallel program will spend a considerable amount of time waiting for locks on data structures it needs to continue executing, or will it using some form of I/O device which are inherently slower than reading or writing to the RAM or cache of the CPU. When this is the case, the thread might require very little or no CPU time at all, perhaps leaving one of the logical cores running under full capacity. So whenever a program has a lot of synchronization of access to outside I/O devices, one should strongly consider using a few more threads than there are logical cores, to ensure that each core will always be running at full capacity.

Note that if the program is running on a machine not dedicated to only run that program, there will almost certainly be a large number of other processes running side-by-side with the program, and each of the process will also use a large number of threads. Typically these processes will not be doing any CPU intensive work very frequently, therefore the time they are allocated by the scheduler will be minimal and their effect on the performance on your program will also be minimal.

Another issue one must consider is whether or not all threads are given small or large amounts of work. Due to the cost of creating and later tearing down a thread, the amount of work done by a thread should be substantial enough to justify that cost, otherwise making the thread will slow the program down. If the problem at hand only consists of a lot of small operations, one should not immediately dismiss parallelization as a valuable tool to increases performance. Instead of executing each small operation on a separate thread, one could make batches of small operations

and then have a thread execute many operations; these batches would again have to be substantial enough to justify the creation of a new thread. This can greatly lower the needed for creating new threads, as much fewer will be needed in order to execute all the operations. This approach has the downside, that you may not be sure that all of the created batches will take an equal amount of time to execute, meaning that some threads will finish before others, leaving the CPU running at less than full capacity. Instead one might want to find a more dynamic solution to the problem. One way of providing such a solution, would be to keep a centralized list of operations that have yet to be executed, then each thread would be able to an operation out of the list and execute. All the threads would continue doing so, until no elements in the list remains. This provides a much more dynamic solution, where each of the operations is allowed to consume greatly varying amounts of time and the CPU is better utilized. Of course, some heuristic is required to ensure that the most time consuming operations are not executed last, as this again would lead to several threads having nothing left to do and some threads having a lot of work left on execution of their operation. Dealing with the problem of uneven distribution of workloads on the cores is referred to as load balancing.

2.6.3 Increased complexity of implementation

Even when implementing simple algorithms, attempts to parallelize the execution can greatly increases the complexity of actually implementing the algorithm as numerous extra considerations must be taken to ensure the correctness of the algorithm and ensure an increase in execution speed. Another issue also arises when race conditions are accidentally implemented into the algorithm, namely removing them. Traditional debugging will often be unable to show you where the race condition is, as most debuggers are not well equipped to debug a program running on multiple threads. In most cases, a debugger will only pause the execution of the first thread that hits a breakpoint, leaving the remaining threads still running and possibly changing the data of the thread the debugger paused. This makes stepping through a small piece of parallel code much less rewarding in terms of information than a similar piece of serialized code would.

2.7 Parallelization in .NET

2.7.1 Tasks

Tasks are the preferred way to represent an asynchronous operation in the .NET framework. Tasks operate at a much higher abstraction level than threads do, as creating a task will not always result in the creation of a thread and it is not possible to specify which core a task is executed on. As such, tasks are a very high level construct which simply contains a piece code you may wish to run asynchronously.

Tasks support the most important operations used to control the flow of a parallel program, such as various wait functions, the simplest of which causes the calling thread to wait for the task to finish execution before continuing. Two options are also given for starting a task, one is to start asynchronously, where it will run in parallel with the calling thread and the other is to start it synchronously, which causes the calling thread to start running the task. When a task is started, it will be queued on the current Thread Pool (See 2.7.2), and in most cases start executing shortly after on a thread selected by the thread pool. Due to the abstraction level of tasks, not all of the control available to threads is available to tasks, since a lot of the scheduling and thread creation and removal is handled by various heuristics in the .NET framework. But using tasks over threads is often simpler and it is often faster as well, due to the way the thread pool optimizes its use of threads.

The following illustration shows how tasks are not directly connected to threads, and we see that all scheduling of tasks on threads is handled by the Task Parallel Library.

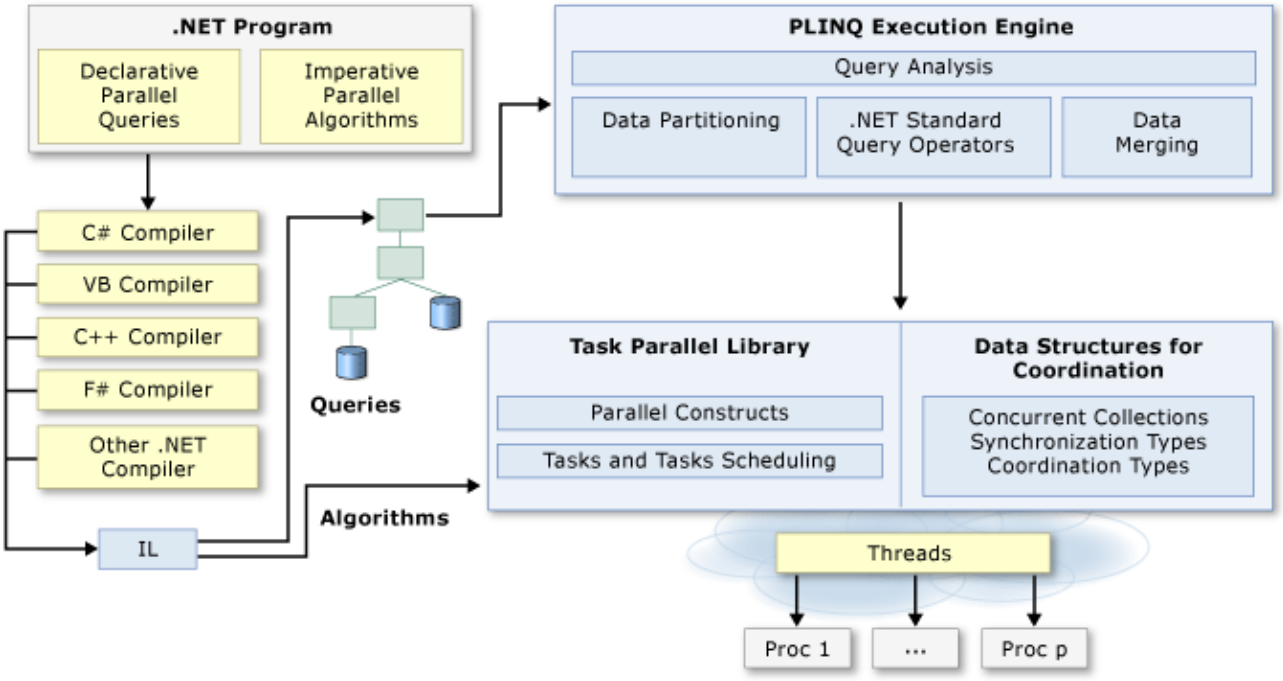


Figure 2 Showing how the .NET framework handles parallelization when using tasks.[6]

2.7.2 Thread Pool

The basis for the thread pool is fairly simple, since it is what the name says, a pool of threads. When an application is started, a number of threads are injected into the thread pool, this number is usually the number of cores available on the machine, and put to sleep. When a task is started, the thread pool may wake one of its threads and start executing the task on that thread, this helps mitigate the overhead incurred by creating threads, as the threads had been created previously. When the task finishes, the thread is simply put back to sleep. This is the basic principle of the thread pool though a lot of heuristics are used to optimize how it uses the threads and it may even add more if necessary. It is important to note that all threads in the thread pool is considered background threads, and this type of thread will not keep a program running after the foreground thread has finished, so if the main thread finishes the program will terminate regardless of all background threads were doing work or not.

[6] http://msdn.microsoft.com/en-us/library/dd460693.aspx

Another way to use the threads of the thread pool is to use the method QueueUserWorkItem, which takes a method as argument and runs it on the next available thread. This process bypasses the task creation process, which means it has much lower overhead but it does come with any sort of control, this means that possible synchronization must enforced in some other way. The framework has several constructs for this.

2.7.3 Parallel loops

The .NET Framework also contains parallel versions of the standard loop constructs, the for-loop and the forEach-loop. Theses parallel loops are well suited for when and algorithm has a lot of work to do, with shared data access, as the implementation of the loops allows them to load balance efficiently and minimize the overhead incurred by the tasks given to the thread pool. When iterating over a list, in order to minimize the overhead and improve load balancing, it supports an operation for splitting the list it is iterating over into smaller chunks, which is then passed on to a task. These chunks will vary in size over the execution, or the programmer can set a fixed limit, in order to find the best possible size. It also tries to maximize the utilization of the cache on the CPU, by trying to have the tasks work in the same area of the collection, so most of the threads will be able to access the data they need directly in the CPU cache. They also support breaking the loop early, as their serial versions do. Implementing these loops is slightly different than the serial loops, as these are in fact just methods whose arguments look like that of a normal loop.

Here is a very simple example of a parallel forEach loop and its serial counterpart:

```
Parallel.ForEach(list, item => testMethod(item));

foreach (var item in list)
{
     testMethod(item);
}
```

Both these loops do the same work, the parallel version simply attempts to speed up the process as much as possible, by using all available cores to do so. The forEach loop allows for much more complex behavior than this example shows such as specifying how much parallelism it may utilize and the exact way it partitions the list into chunks.

With all of these optimizations, this parallel version of the forEach loop and the for loop, should be used whenever possible to maximize the usage of your CPU and to minimized the time required to implement the desired logic.

2.7.4 Locks and monitors

The basic construct for mutual exclusion in the .NET Framework is the Monitor. It has two key methods, Enter and Exit, where enter takes an object as parameter and acquires the mutual exclusion lock on that object, and Exit releases the lock. This is used to ensure mutual exclusion for some fragment of code located between an Enter and Exit statement. The object passed as parameter should be a private object created specifically for the purpose of being used as the locking object. Using value-types will cause exceptions and using publicly available objects can make it much harder to avoid deadlocks, since an outside programmer may not know which object is being used to acquire locks and accidentally use the same object for another unrelated lock. It is important that Exit is always called following an Enter call, despite what happens inside the mutually exclusive code, this is done by wrapping the code and the Exit statement in a try…finally construct as such:

```
private object objectLock = new object();

…

Monitor.Enter(objectLock);
try
{
    //Code that requires mutual exclusion.
}
finally
{
    Monitor.Exit(objectLock);
}
```

This code represent the correct way of using the Enter and Exit statements, even if the code in the try part fails with an exception, the lock is still released as it should be since we left that part that required mutual exclusion.

The monitor class also has support for Wait, Pulse and PulseAll calls. Where the wait call releases the lock and waits to be notified by some future call to Pulse or PulseAll. This can be used to achieve synchronization and optimize access to objects that can cause side effects.

The correct way to use the Enter and Exit statements may not be used by a developer, as you can easily neglect the try…finally construct or forget to Exit after an Enter call, and both of these will only result in problems at runtime. To make the process simpler, the .NET Frame has a Lock keyword in C# (other keywords are used in other .NET languages, but have the same effect), which is simpler way of writing the correct Enter and Exit statements, it looks like this:

```
private object objectLock = new object();

…

lock (objectLock)
{
    //Code that requires mutual exclusion.
}
```

In this case, the developer environment will inform you of a missing end bracket at compile time and the lock will always be released, regardless of exceptions throwing during execution of the mutually exclusive code. The two code fragments do the exact same thing, but using the Lock keyword is simpler and much less prone to be incorrect.

2.7.5 Atomic Actions

To avoid race conditions associated with certain operations, such as incrementing counters, a lock could be used to ensure mutually exclusive access to the counter. These race conditions appear since incrementing a counter is not, usually, an atomic actions which means increments might be lost due to caching by the CPU. This constant locking can be slow if the value needs to be updated

frequently. Instead of using locks in such a manner, the .NET Framework provides a convenient way of performing very basic actions in an atomic fashion. The Interlocked class contains several static methods, such as increment and add, which can be used to perform those actions atomically, eliminating the need for using locks everywhere.

3 Parallelization of Backtrack Search

3.1 Tree Traversal

Some problems can be modeled as a tree traversal problem. For instance SAT solving or the N-Queen problem can both be modeled as a tree where nodes represent a decision and a path is simply a list of decisions, and a solution is a path that fulfills a certain criteria or a lack of such a path to prove a lack of a solution. In order to conclusively say that a problem has no solution, all possibilities must have been checked by some traversal or eliminated by some heuristic. Since we are traversing a tree, it is clear that all paths used to traverse the entire tree will be unique, unless the same path is traversed twice which serves no purpose, and traversing the tree does not cause any changes to it. This is a clear example of a delightfully parallel problem, as it has no side effects and a natural division of the problem into sub-problems exists.

All of these traversals may be done completely independent of each other and therefore also completely asynchronously[7]. This means that an algorithm for doing such traversals have virtually no serial part, and looking at Amdahl's Law, this allows us to gain a speedup roughly equal to the number of cores without ever being limited by the serial portion. Tree traversal is therefore a very good problem to try and parallelize, as the speedup can be immense.

[7] If something is asynchronous, it can happen independently of the main program.

3.2 Backtrack Search

Backtrack search is a general approach to find all solutions for a given problem, or a single one by terminating once a solution has been found. It works by continuously building up a list of candidates, in a graph a candidate would be a node, and then removing a candidate once it realizes that it could never be part of a solution and then trying some other candidate. The removal of a candidate will often remove a large set of other possible candidates as well, which helps to quickly reduce the problem to a more manageable size. The classic example of backtrack search is the 8-Queen problem, or its more general form the N-Queen problem, in both of which a solution consists of a placement of all queens, on a chess board where each side has the length of the number of queens, such that no queen can capture another queen. Other examples problems that may be solved using the backtrack search algorithm includes Sudoku and SAT solving.

The general backtrack search algorithm implicitly constructs a tree during its execution, where each node corresponds to a candidate and the path it has traversed contains the list of candidates. In the case of the N-Queen problem, the tree is formed by having each position on chess board represent a candidate and such a candidate will be connected to all other candidates if their position is free and once it is considered safe. A candidate will only be accepted, considered safe, if that position is safe from all other queens on the table. A queen can move horizontally, vertically and diagonally to capture another queen, therefore any position where a queen can move to is considered unsafe in the case of the N-Queen problem. A solution is said to be found once the list of candidates contains exactly N candidates, which is N placements of queens, since the algorithm will never accept a candidate if it cannot be part of a solution.

The general algorithm looks as follows:

```
BacktrackSearch(P,C) =
  If !Safe(P,C) then return
  R = P.Add(C)
  if Complete(R) then Output(R)
  S = first(R,C)
  While S != null
      BacktrackSearch(R,S)
      S = next(R,S)
  Return
```

Where P is the partial solution and C is a candidate.

- Safe checks if C can extend the partial solution P, and returns in the case that it cannot, otherwise C extends P.
- Complete checks if the new partial solution R is a solution and then uses Output to return the solution.
- First finds the first possible candidate S that may extend R.
- BacktrackSearch is then recursively called with the updated solution R and the new candidate S.
- If the recursive call returns, S is updated to be the next possible extension. This is done until there are no more possible extensions, and the method returns.

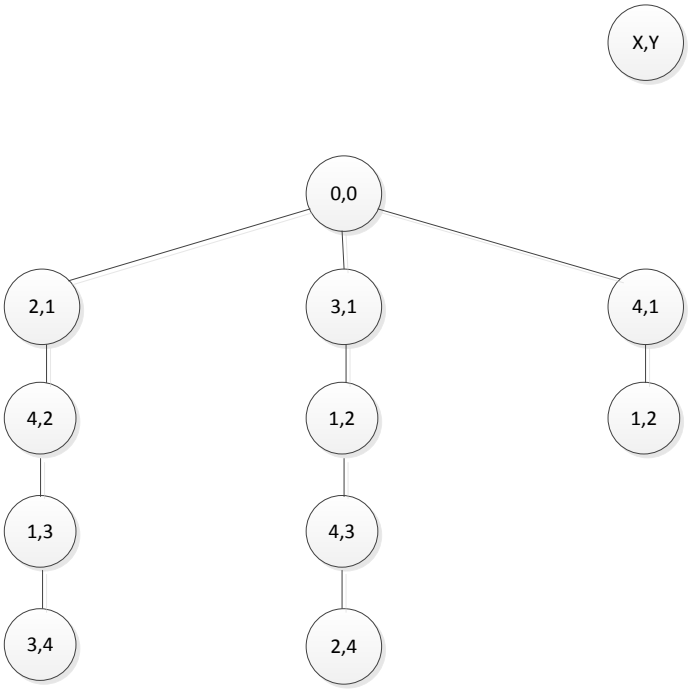
For the N-Queen problem, the first queen would be placed on row zero, and the first and next calls would return the positions of row one. It continues to place a queen on the next row until all rows have a queen or no queen can be placed on the next row and it backtracks. The actual implementation of the safe and the complete method will of course vary greatly with the problem at hand.

In order to make the parallelization of the algorithm easier, the actual implementation follows this slightly modified version of the algorithm:

```
BacktrackSearch(P,C) =
  If !Safe(P,C) then return
  R = P.Add(C)
  if Complete(R) then Output(R)
  nexts = getNexts(R,C)
  For each element S in nexts
      BacktrackSearch(R,S)
  Return
```

The getNexts method returns all safe positions of the row above R. This allows us to get all the positions we need to check, instead of getting them one at a time, which will make it easier to divide the work between Tasks used in the parallelization.

This algorithm can be viewed as traversing a tree that, in the case of N = 5, will look partially like this:



Note that this is only the tree originating from (0, 0), all the other positions of row zero would also have a tree similar to this. All these trees are tied together at a top node that is not part of the actual solution, but merely a result of the loop that loops over all positions of row zero.

3.3 Finding all solutions

Finding all solutions means finding all the paths that satisfy our criteria of having length N. The simplest way to parallelize this would be to start a new thread for each of the nodes in row zero, let them all finish with their part of the search tree and gather up the solutions they came across. This is a very good approach to the problem, but it requires some modifications before it can run optimally. For a moderately sized problem, say N>16, this solution will most likely produce more threads than there are logical cores on the computer leading to oversubscription which will slow down the execution.

Instead, the positions of row zero could be broken in to a number of small groups equal to the number of logical cores available for use; this would solve the problem of oversubscription but potentially lead to a load imbalance in the case where one of the groups finished much faster than the slowest group. To handle the load imbalance, some sort of queue of positions would have to be implemented and each thread could take the first element of the queue and start the search from there. Once a thread finished its search, it would take another element out of the queue and start again. This will give us a better load balance, but the slowest start element might be the very last in the queue, potentially giving us the same problem as originally. To properly deal with load imbalance, some heuristic must be applied to the queue in order to sort it from slowest to fastest, so the slowest start element will be taken first, this gives the program the best chance finishing all the searches at about the same time.

Another approach would be to allow threads to help out each other, by segmenting the possible choices of positions based on an estimate of how many solutions a given position would have. More tasks would then be allocated to the segments where there may be many solutions, and fewer tasks for the segments with fewer solutions. This can quickly become very complex to

implement, and before starting on such a task, the problem should be analyzed to see if this kind of optimization is even warranted. In the case of the N-Queen problem, once a reasonable size of N is reached, about N>12, most of the positions at row zero will contain roughly the same amount of solutions and take equal time to find. And when N is still small, the time it takes to find all solutions, is so short, much less than a second, that the optimizations would just slow it down.

The .NET Framework contains a construct that will allow us to parallelize at row zero, while avoiding over subscription and minimizing load imbalances, namely the parallel forEach loop. This efficiently handles the problems outlined above and is very simple to implement. The full code for parallelizing the backtrack search for finding all solution is as follows:

```
var candidateList = GetStartCandidates(dataGrid);
Parallel.ForEach(candidateList, s => SolveAll(dataGrid, s));
```

Where the candidate list contains some positions of row zero (See 4.1 for explanation), and the SolveAll method is simply the general backtrack algorithm described in 3.2. The dataGrid is the partial solution, initially containing no nodes, and s is a candidate.

3.4 Finding one solution

When looking for a single solution, we want to invest all resources into finding any single solution as fast as possible and then terminate the execution. This means we are interested in exploring single paths as fast as possible, which leaves out the solution used for finding all paths. The general approach will be one similar to that of the famous depth-first search, as this employs the tactic of going deep in to a single path. Using the N-Queen problem as an example for parallelizing the general backtrack search, a series of solutions will be explored. For small problems, finding a single solution, even with a serial version of the algorithm, takes very little time. For the problem to benefit from parallelization the input size should be larger than fifteen to twenty depending on the speed of the platform.

A simple way of exploring a path starting at some position at row zero, is to place a queen at that position and then do the exact same as finding all solutions but starting out with a partial solution consisting of that one queen placement and the list of candidates would contain the nodes of row one. This will find a single solution faster than a serial version of the algorithm would, but not much faster. The speed of the parallel version with an input size of N, would be comparable to the serial version with input size of N-1, assuming the first queen placement is part of at least one solution which it will be in reasonably sized problems, as placing one queen effectively reduces the size of the problem by about one. The problem with this is that each thread will still have large portion of the search to check.

Another approach is to split the path into two new paths, each time a queen is placed. The idea is that some tasks starts with a single queen placement at row zero and before it places a queen at row one, it splits the list of candidates into two equally sized parts then creates a new task and passes one part of the list to it. Then the originally task would continue in one direction and the new task would continue in another, effectively splitting the path into two.

But this continuous split of paths will very quickly lead to oversubscription, as there would be 2^L threads total when the algorithm reached level L. The over subscription can easily be dealt with by introducing a counter for the number of tasks currently being used, and simply preventing the creation of tasks if there as many tasks as there are logical cores. Such a counter would have to be both incremented and decremented atomically, to ensure that count stays correct, which can be done by using the Interlocked class in the .NET Framework. Load balancing is not much of a problem, since each tasks will very often try to create a new task, and will be allowed to do so the instant some task finishes its assigned workload, leaving all logical cores running all the time.

There is another problem with this approach; a new task may be created at the N-1 level, where all it has to check are the candidates at level N which there can never be more than one of. This is too little work to warrant the creation of a new task, and this will also be the case even at lower levels of the tree. To correct this problem, a variable must be introduced, specifying the maximum level new tasks may be created at. This helps ensure that all tasks will have a large enough amount of work to do, so they are worth creating.

Along with this idea of a maximum depth to create new tasks, we also need a minimum depth to create a task. When the problem becomes bigger, the choices of the first few queen positions are trivial as they will almost always be part of some solution. If we use all our available tasks for checking paths that we are sure will lead to a solution, we might as well just use a single task and still achieve the same speed. The minimum depth, if set correctly, can help greatly reduce the size of the problem allowing for speed ups higher than the number of cores on the platform, as we are effectively removing the first several choices since they are trivial.

When we increase the minimum depth, the number of solutions that can be found using the queens positions before the minimum depth drops drastically. But with more cores available for the execution of the algorithm, the higher the minimum depth can be set, since all we need to find is a single solution so we do not care if many of the cores were assigned a path that may never lead to a solution.

The difference between the maximum and minimum depth must be large enough such that there will be created as many threads as specified, if the difference is too small, the splitting behavior will not have enough room to start all the tasks. It is also important to ensure that the difference is not too big, as this would allow the splitting behavior to create too many threads.

4 Optimization Experiments in N-Queen problem

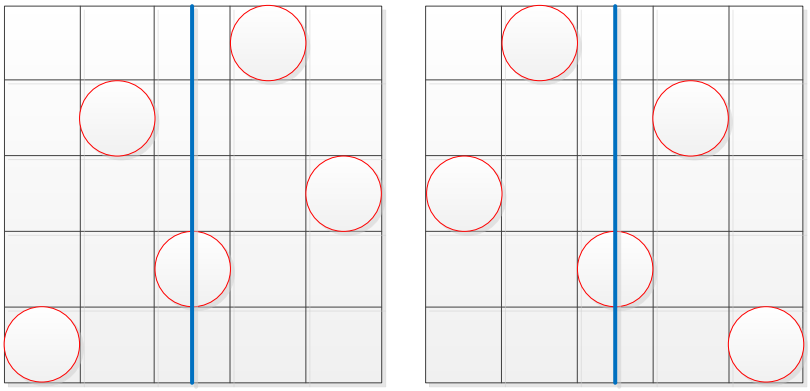
4.1 Exploiting Symmetry

To increase the speed of finding all solutions, the symmetry of the chess board can be used to generate new solutions based on old solutions, simply by mirroring around the x or y-axis.

To find all solutions, the initial list of candidates contains all positions in row zero. We can easily see that all solutions starting with the first position in row zero (0,0) can be mirrored on the y-axis to give all solutions for the last position in row zero (N,0). If we use an initial list containing only

one half of row zero, we can then mirror all the solutions we find to provide the solutions for the other half of row zero. Here is an example of how a y-axis mirroring provides an additional solution.

Y-Axis mirroring

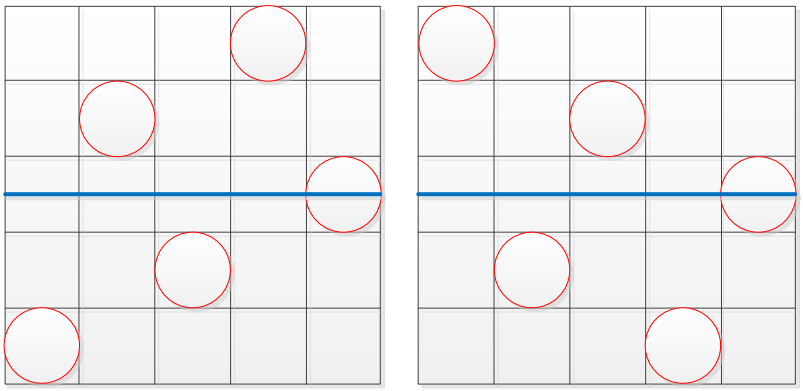


Using the x-axis to generate new solutions may seem possible at first, but since the initial selection of candidates is given by a row rather than a column, it turns out the x-axis cannot provide any new solutions. If we have found solutions of one half of row zero and mirrored all those, we have all solutions, since a solution must contain exactly one position in each row and we have all solutions containing the positions in row zero, we must have all solutions. Otherwise, a solution must exist which contains no positions of row zero, but this cannot be a solution due to the definition.

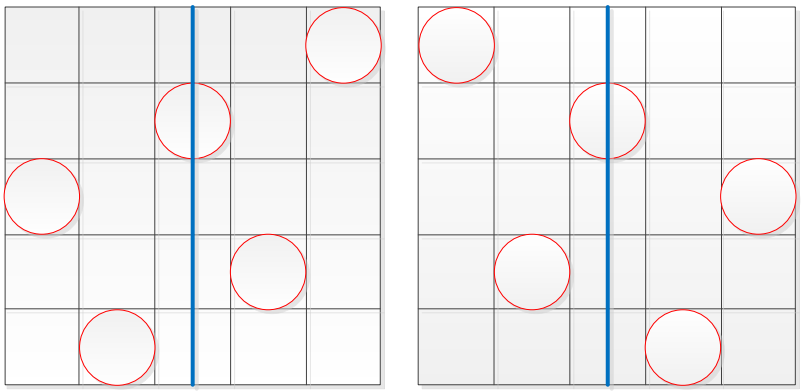
To generate anything new from an x-axis reflection, this must happen prior to the y-axis reflection and after the solutions of half of row zero is found, else there are no solutions to mirror. If a solution contained in a position of row zero is reflected across the y-axis, another solution will appear which also contains a position P in row zero. The position P may either be part of the half of row zero chosen for the initial candidate list or it may not, in the case where it is part of it, P is not a new solution, otherwise it is a new solution. All the new solutions an x-axis reflection could generate can also be generated by the y-axis reflection, since they must all contain a position of row zero. This would imply that you could simply do either of the reflections, but this is not the case either, as the reflections gained from the x-axis is not guaranteed to all be new, and if just a single one of those reflections are not new, we will not have all solutions. This is because we know there should be twice as many solutions as found we using only half of the first row (if N is odd,

there would be 2 times the number of solutions found in all but the middle position), and we would then be missing at least one of these. Here is an example showing how an x-axis mirroring may give the same solution as some other y-axis mirroring would.

X-Axis mirroring



Y-Axis mirroring providing the same solution.



The only way to use the x-axis for generating new solutions is to have the initial list of candidates contain half of column zero, rather than row zero, but then the y-axis is useless.

4.2 Randomization

The algorithm used for finding solutions will mostly always traverse the tree in the same way, this means it may run in to the same dead ends every time and have to backtrack or it may avoid the same dead ends and find a solution fast, all depending on the input size. This behavior only

happens with multiple runs of the same input size, not in a single run. Since there are more incorrect solutions than there are correct ones, it would stand to reason that it will make the same mistakes more often than it will make the same correct choices. To try and avoid this problem, some randomization could be added to the part of the algorithm that determines which position is the next to try, this would lead to a much more varying runtime of the algorithm but hopefully it would be faster on average.

However, there is one detail about the non-randomized algorithm that makes the randomized version much slower if it were to be implemented; the normal algorithm implicitly stores data about paths it has already traversed, since the approach is systematic in the way the next position is chosen, it will never attempt to traverse the same path twice or make the same mistake twice, on a single run. This implicit data comes from the systematic approach, since it will use a list of positions and simply go through that list and it will never take a position it has already used in the list, but it will take the next element or get a list of positions for the next row/column. The randomized algorithm would not implicitly have this information, but the path would have to be stored whenever a mistake or a solution was found, and other traversals would have to check if they are on the path of something that has been proven to work or not to work. The data structure and associated functions required to provide such information would cause a considerable amount of overhead, which the normal algorithm can completely avoid, leading to the idea of randomization being abandoned.

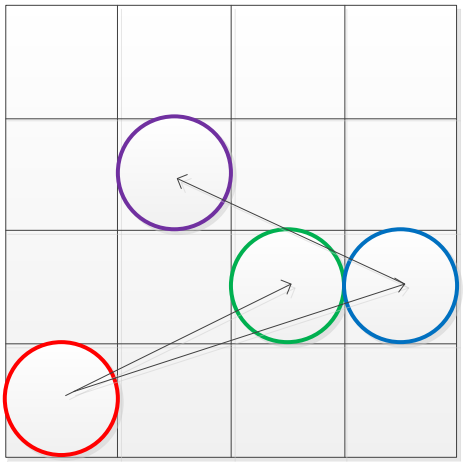
4.3 Using Counter Examples

4.3.1 Counter examples

When a path turns out to be a dead end, the algorithm has no candidates which may extend it, forcing the algorithm to backtrack, that path can be said to be a counter example for any proposed solutions containing it. Such a counter example could also be used during the construction of a path, if a partial path contained the counter example the algorithm should backtrack. If a partial

path contains part of a counter example, the order of the next positions to check may be altered in such a way that the position that is also contained in the counter example is tested last or not at all if that position is the final position in the counter example. Since a counter example is found every time the algorithm backtracks, this gives us a lot of counter example, many of which may be sub-sets of each other, so we introduce the notion of a shortest possible counter example (SPCE).

Such a counter example could contain many other counter examples, but only the shortest one would have to be stored as it is simply a more general counter example than the ones it contain. When a counter example is found and the algorithm backtracks, it may either backtrack again since the first backtrack ended up in yet another counter example, or it may try different positions. If all these different positions also end up being a counter example, another backtrack is performed, the algorithm may continue this behavior until at some point, a different position leads to an actual solution and then the shortest possible counter example would be the one that we had just before the last backtrack happened. So given many backtracks, the SPCE would be found at the very last one of them. It would contain all the counter examples contained in the other backtracks before, and therefore be the shortest and most general backtrack possible. Many such SPCEs may exist, since not all solution start at the same node and in the worst case, each of the positions of row zero could be a SPCE, giving us N SPCEs, this happens in the case where no solutions exists for the problem of size N.



In this example we see 1 SPCE and 3 CEs. The red and the green position make up one CE and the red and the blue positions make up another and finally the red, blue and purple make up the third.

But we see that these three CEs all share the red position, and no solutions may be found starting from the red position, making the red position a SPCE.

One way of using these counter examples would be to store them whenever we find one in a given run of the algorithm, and use it for the remainder of the run. However, using the same reasoning about the systematic approach as in the randomization, we can quickly realize that if we find a counter example in the problem we are currently trying to solve, we can be sure that we will not hit it again in the same run of the algorithm, due to the how the next positions are chosen. This means that any counter examples found during a search for solutions cannot be used.

The only way to use counter examples in a search for solutions, is to have the counter examples prior to beginning the search, as we would then be able to use some heuristic which allows us to avoid them when searching. To find the counter examples of a problem of size N, we will have to find the solutions as well, otherwise we cannot be sure that a SPCE is really the shortest possible or even a counter example at all.

When a backtrack happens we know for a fact it was because of a dead end and this gives us an actual counter example. If we were to simply store all counter examples found in such a way, the structure containing these would become immense and using it later on to check if some path is a partial path of one of these counter examples would take far too long to check.

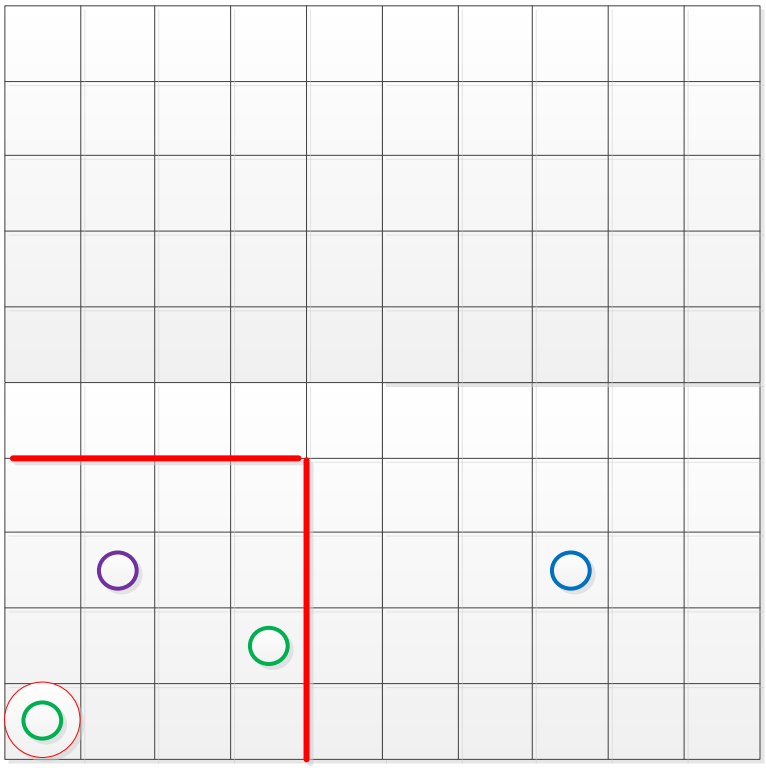
So instead of trying to find SPCEs in a problem of the same size as the one we want solutions for, we may try finding them in problems that are smaller.

4.3.2 Counter examples from smaller problems

The obvious advantage of using smaller problems to find SPCEs in is that smaller problems run much faster due to the exponential running time of the algorithm, however, any SPCEs found for smaller problems also provide much less information since they are not actual counter examples for full size problem. Therefore they cannot eliminate any paths from the full size problem, but

they might be able to influence the choice of paths made in the full problem in a way that speeds up the run time.

To use a SPCE from a smaller problem in the full size problem, we have to realize what information can be gained from it. Placing a queen in the full sized problem N may tell us that we are working towards constructing a path that was a counter example in the smaller problem M, obviously if we place a queen in such a way that the path would no longer fit inside M, the counter example could no longer be used. This is what we will try to use, when we realize we are on a path that was a counter example in some smaller problem, we try to place the next queen in a way that breaks that counter example, in the hopes that such an approach will help us avoid backtracks.



In the image we see the red circle is a SPCE of a problem of size four, outlined with red, and the green circles are queen placements in the full size problem, size ten. We clearly see that the green circles make up a path that was considered a counter example in size four, and placing the next green circle on the purple circle would simply extend such a counter whereas placing it on the blue would break it.

We need more than a single counter example for a single size if we hope to gain anything, so we go looking for SPCEs in many differently sized smaller problems and store these in a quickly accessible fashion. Since none of these SPCEs can eliminate any positions, the heuristic simply becomes an ordering of the choices we have for placing queens on each row, where we prefer to place queens in such a way that we avoid the smaller SPCEs.

4.4 Check for invalid rows

We know from the definition of a solution, that each row must contain a queen or it cannot be a solution. This information may be used to stop the exploration of a giving path prematurely, in the case where some row that does not yet have a queen has no valid positions in it, because this means that the path can never lead to a solution. Blindly checking all the rows that does not yet contain a queen for this criteria would be too slow, as the method for checking if a single position is safe runs in O(N) and there may be N^2 positions to check.

Instead we should choose to only check a single row. Checking the row we are about to try to place a queen on is not needed as already happens due to the structure of the algorithm and if there are no valid positions to place a new queen the algorithm backtracks. Checking any other row than just the next one can only yield minimal benefits; the added runtime caused by such a check would remove any such benefits, since it takes a lot of queen placements to completely cover all positions in a given row.

Placing a queen will at most remove three possible positions on some row; one position from vertical movement and two from diagonal movement, this would mean we have to place at least N/3 queens before any row can be invalid. But a queen may only remove from rows that are within N/2 rows of the queen. A queen placement may also only remove one, two or even no positions, placing queen that remove three positions at a given row can only be done very few times, since any additional placements will start removing some of the same positions again. It turns out you have to place at least N/2 queens to invalidate a single row. These N/2 queens would have to be place in a way that removes the most positions from a row, but such an

approach will never lead to a solution, and for the approach used by the algorithm, starting at row zero and working towards row N, many more placements are needed to invalidate some row.

It turns out that checking any other row provides no real benefit, as by the time such a row becomes invalid, only a few queens will need to be placed for the check of the next row to reach the same conclusion, or in most cases, no queens at all, this is simply because of the way the algorithm places queens, going from row zero to row N. To see a small benefit from such a check, the check should only start happening once enough queens have been placed for there to be a real chance of some row actually being invalid.

5 Architecture of the implementation

5.1 Data grid

The data grid is the data structure used to store queen placements. Given an x-y coordinate for the chess board, it can tell you if a queen is at that position or not, or the full list of queen placements can be retrieved. See appendix 1 for the full code.

The internal structure of the data grid used to store the queen positions have gone through several iterations, the first being a simple two-dimensional list where each element had a Boolean value indicating if a queen was placed there or not. This approach offers simplicity in checking individual positions, as it allows two-dimensional indexing out of the box; however, it contains N^2 elements giving it poor size scaling. The .NET Framework has a data structure specifically for the purpose of storing an array of Booleans, the BitArray class and this was used in another iteration of the internal structure. Since the BitArray is one dimensional, this worked as a flattened version of the two-dimensional list structure. This meant that an x-y coordinate would have to be translated into a single value, to allow lookup in the BitArray and this adds a slight bit of overhead compared to the normal indexing function. The BitArray also contained N^2 elements, but in this

case, each element only takes up a single bit instead of a full byte as a Boolean normally does. This still gives us poor size scaling, but with a much lower constant.

The final version of the data grid uses a single list of positions, where each position represents a queen placement. This data structure takes advantage of the fact that the algorithm places queens in a systematic way, such that a new queen will only be placed one row above the previous one. When a lookup is performed on this data structure, the y-coordinate of the position is used as an index into the list retrieving an element E, a check first ensures that such an index actually exists, and then the x-coordinate from the indexer is compared to the x-coordinate stored in the element E, if the x values match, it returns true and otherwise false. This structure will only contain up to N elements, giving us much better size scaling and the indexing function also runs fast as it only compares to values in addition to the lookup itself. It does limit how the algorithm places it queens, as this structure would not work if queens are not placed the way they are, as the y-coordinate used to retrieve element E would return the wrong element E, since the E y-coordinate would not match the y-coordinate from the index. Both the x- and y-coordinates are stored in the list, despite the y-coordinate being redundant when just performing lookups but it is useful when getting the full list of positions.

The data grid can also return a version of itself where every queen position is mirrored around the y-axis.

The data grid is immutable, at least when seen from outside the class, so when a new queen is placed, the data grid returns a copy of itself with the one queen being added. This immutability helps the parallelization, as it eliminates the concern of shared data access when dealing with the data grid.

A non-immutable version the data grid also exists, but this was abandoned early as it provided no measureable speed benefit, and the total size of the data stays very limited so the immutable approach does not cause any out of memory problems.

5.2 Solver

The solver is the core of the program; it contains all the methods and algorithms needed to solve the problem with and without parallelization. The solver contains four segments of methods, one for finding all solutions, one for finding a single solution, one for finding SPCEs and a bunch of helper methods including the methods checking if a position is considered safe. Each of the segments for finding solutions or SPCEs use the same basic backtrack search algorithm, but modified to best suit the need of the particular job. See appendix 2 for the full code.

The implementation for finding all solutions is the simplest of the three, as it is the basic backtrack search algorithm initialized in a specific way. Half the start candidates for the problem is retrieved, this list is candidates is the iterated over using the Parallel.ForEach construct with each iteration starting a new backtrack search running on a new task. As noted earlier, this construct will avoid oversubscription and otherwise optimize each task, making it the ideal choice for finding all solution. There are no race conditions in this implementation, as each task has a completely separate part of the problem. When a task finds a solution, the solution is then added to a list of solutions and the algorithm backtracks as it would if it hit a dead end, allowing it to continue looking for more solutions until it has found all possible solutions starting with the start candidate it was originally given. Once all the solutions have been found for the given start candidates, we have to mirror all of them, since we only initialized the algorithm with half the positions of row zero, to get the remaining solutions.

The implementation for finding a single solution functions in much the same way as the one for finding all solutions, since it is still the same algorithm being used. The parallelization instead occurs inside the method and only when specific conditions are fulfilled and this version of the algorithm will terminate once a single solution has been found. To decide if a new Task should be created, it checks if there already exists too many tasks such that adding another will bring to total number of tasks created above some set threshold. It then checks if the current depth of the tree is between the minimum and maximum depths specified. Both of these parameters may be set on the Solver object. Lastly, we make sure that the list of valid position “nexts” contains more than one position.

If all of these checks are passed, we begin the process of creating a new task, first by splitting the list of valid positions in two, one half goes to the new task and the other half is left for the old task. The list of valid positions may be sorted before the split based on the information in the Counter Example Set, if that option is selected; the split does not split the list in the middle, but takes every other element out. We then atomically increment the Task counter, and then finally create the new Task, telling it to run a special method created for this purpose. Methods used for Tasks must take exactly one object, containing all the relevant data which in this case is half the valid positions and a copy of the current partial solution, as parameter or no parameters at all and this method then simply calls the algorithm. When the call to the algorithm finishes, the Task counter is atomically decremented with one. If at any point a solution is found, a Boolean is so to true so when all other threads checks this Boolean they will stop their execution.

The final use of the algorithm is for finding SPCEs. This works like the implementation for finding all solutions, except for a few important differences. The first difference is that you specify a range of problem sizes, rather than just a single size, and SPCEs will be found in all of these and added to the Counter Example Set. It also contains the logic for finding SPCEs in the way described in Counter examples from smaller problems.

5.3 Counter Example Set

This class is used for storing and retrieving SPCEs found by the Solver. The base data structure for storing these SPCEs is a dictionary with positions as keys and tuples with a dictionary and a list of counter examples as values, the dictionary in the tuples are constructed in the same way, giving us a recursive data structure. See appendix 3 for the full code.

When adding a SPCE to the data structure, the list of queen positions is retrieved from the data grid which is passed as a parameter, and then the dictionary is recursively traversed starting with the first queen position ending with the last queen position. Key/Value pairs are added if a position is not already a key in the relevant dictionary, this helps limit the size of the structure, since we are doing a form of lazy initialization only adding the exact Key/Value pairs we need.

Once the full list of queen positions has been used as a lookup, the list of counter examples found in the tuple is updated with the counter examples contained in the data grid. In end result is a data structure that functions like a dictionary with lists of positions as keys and lists of counter examples as values would. Since the add function is called by many different threads, it is important that a lock is acquired before any changes to the data structure is allowed.

Retrieving a list of counter examples functions in the same way as adding counter examples, except Key/Value pairs are not added if they are missing, since this simply means that no counter examples exist for the given list of positions. The retrieve function does not use any form of locking to ensure mutually exclusive access, since all changes to the data structure are made before the first retrieve call. We know this because the retrieve function is only used in the function for sorting the list of valid positions in the Solver, and this only happens during the execution of the FindSingleSolution function, which happens after all SPCEs have been found.

For sorting the list of candidate positions, we first check if the current partial solution can be part of any counter examples previously found, by checking the current size of the current data grid against the size of the biggest counter example. The current size of the current data grid is not the size N it was initialized with, but rather the number of queens placed, as we know if more queens have been placed than the largest counter example can hold, then no counter examples found may fit in the current data grid. Next all counter examples the current grid is part of is retrieved and again we check if this list contain any counter examples. We then simply iterate over the list of counter examples and the list of candidate positions, checking if a candidate position extends one of the counter examples or breaks one. If the candidate extends a counter example, a weight of value one is added to it and otherwise nothing is added. Once all counter examples have been checked against all candidates, the list of candidates is sorted with respect to the weight assigned to each candidate and then returned to the solver.

6 Results and discussion

6.1 Result gathering

All the results were gathered using a debugging interface created for this project, see appendix 4 and 14 for the implementation of the interface and a screenshot. For each test, different settings were used and the settings used will be noted next to each of the test results so that any results may be reproduced at a later time. All the tests were run on a Windows 7 platform with four physical cores with Hyper Threading enabled, each running 2.8 GHz.

In the test results, speedup refers to how many times faster the parallel or optimized version was compared to the serial or non-optimized version. A speedup of less than one means it was actually slowed down.

6.2 Results of parallelization

6.2.1 Find all solutions

Relevant settings used were: Optimize = false, FindAll = true.

To make the interface use the Parallel.ForEach loop when finding all solutions, a number greater than zero has to be entered in the textbox marked “threads”, a value of zero forces the use of a standard ForEach loop. MinDepth and MaxDepth have no influence in this test. In all cases, both loops found the same number of solutions.

	Parallel.ForEach	Standard ForEach	Speedup
N = 8	0,035s	0,026s	0,74
N = 10	0,136s	0,179s	1,32
N = 12	1,407s	5,387s	3,83
N = 13	8,318s	32,520s	3,91
N = 14	53,205s	211,742s	3,98

We see that for small problems, the overhead incurred by using multiple cores becomes so high relative to the time it takes to solve the problem that it slows down the algorithm. Once the problem size get bigger, we approach a speedup of four, whereas one might have expected a speedup closer to eight, given that the CPU has eight logical cores. But these extra logical cores provided by Hyper Threading may only give a modest performance increase[8]. This means we are only interested in the number of physical cores present on the CPU, in which case the speedup seen in the tests are what one should expect of a CPU with four physical cores.

This speedup would also be expected according to Amdahl's Law, as the only portion of the program that has to be run serially is adding a solution to the list of solution, which is a very fast operation that runs in O(1). This means that the portion of the program that can be parallel is very close to 100%.

6.2.2 Find single solution

Relevant settings used were: Optimize = false, FindAll = false.

The interface allows changing minimum depth, maximum depth and number of additional tasks. Zero additional tasks means the algorithm will only use a single thread for execution. Minimum and maximum depths have no influence on the algorithm if zero addition tasks are chosen. If a field has a dash in it, it means that the combination of parameters was not tested. Moving to the

[8] See 2.4 Basic CPU Architecture.

right indicates an increase in number of extra tasks used, specified at the top of each column, moving down changes the minimum and maximum depths.

N = 20

Min/Max	0	2	3	4	7
0/3	2,835s	0,029s	0,074s	0,068s	0,031s
1/5	2,831s	0,105s	0,022s	0,041s	0,79s
2/5	-	0,071s	0,075s	0,074s	0,102s

N = 25

Min/Max	0	2	3	4	7
0/3	0,99s	0,441s	0,505s	0,492s	0,503s
2/5	-	0,335s	0,376s	0,396s	0,396s
3/6	-	0,083s	0,089s	0,103s	0,115s

N = 28

Min/Max	0	2	3	4	7
0/3	71,303s	65,543s	70,359s	68,679s	-
3/6	-	0,886s	0,929s	0,923s	0,885s
4/7	-	6,561s	6,496s	6,75s	1,183s
3/7	-	0,831s	0,983s	1,085s	1,555s
4/8	-	6,329s	7,183s	0,813s	1,105s

Looking at the results overall, we see wildly varying speedups from near one going almost up to 100 in the case of N20. Predicting the speedup gained by adding cores using Amdahl's Law is not possible in this case, due to the work assigned to the extra tasks. If the algorithm was run with just a single task, the task might have to explore a large branch of the search tree only to realize the branch did not contain any solutions. Whereas if there were more than one task created, one of the others tasks would start on a separate branch that might lead to a solution without causing a single back track. Obviously, this task that picks the best branch to explore may finish long before

all the other tasks, giving us an unpredictable speed. The speedup gained by parallelizing this algorithm seems to depend on the size of the problem, as changing the size changes where the simplest solution may be found.

We see that increasing the minimum depth on size 20 and 25, does not cause nearly as big a change in execution time as it does for size 28. This difference is again caused by how solutions are placed in the given size, for 20 and 25 there are much fewer solutions left after going three rows into the chess board than compared to size 28. This means that the early choices for queen positions are less trivial for 20 and 25 than for size 28. The maximum depth should stay above the minimum depth, to ensure that the splitting behavior is able to create the full number of cores.

It is clear that the outcome of each test depends on the settings, and tweaking these correctly can give significant difference in execution time, which is clearly demonstrated in N28 where the worst speedup is just above one and the best speedup is around 85.

6.3 Results of optimizations

6.3.1 Symmetry

Relevant settings used were: FindAll = true, tasks > 0. All tests found the same number of solutions.

	Symmetry	No symmetry	Speedup
N = 8	0,007s	0,03s	4,29
N = 10	0,096s	0,178s	1,85
N = 12	2,712	5,395s	1,99

The expected speedup from using symmetry to generate the remaining solutions is just below two. The symmetry should be able to provide us with up to half of the solutions, but mirroring all the solutions takes some time as well, lowering the expected speedup to just below two.

The size 8 test most likely suffers from the variance in execution time the garbage collector in the .NET Framework causes, leading to a much higher than expected speedup, but with such short running time, the variance in time plays too big a role making that result useless.

The two other results have a slightly longer running time, which diminishes the effect the variance has on the test results, and therefore provide more accurate results. We see that both tests come out with a speedup close to the expect two.

6.3.2 Counter examples

Relevant settings used were: FindAll = false, MinDepth = 2, MaxDepth = 5.

This first test was tested using threads = 0.

	Not using SPCEs	Using SPCEs	Speedup
N = 20	2,834s	1,030s	2,75
N = 22	28,251s	1,278s	22,11
N = 25	0,969s	2,505s	0,39
N = 28	71,454s	95,389s	0,75

This second test was tested using threads = 7.

	Not using SPCEs	Using SPCEs	Speedup
N = 20	0,133s	0,759s	0,18
N = 22	0,055s	1,062s	0,05
N = 25	0,373s	2,249s	0,16
N = 28	20,327s	116,367s	0,17

We see that this optimization may provide some speedup compared to a serial version of the algorithm, but it may also slow it down drastically. The speedup comes from choosing a better first

placement of a queen in some cases and most of the slow comes from this first choice being worse. If we look at the algorithm, it counts how many counter examples each candidate position is part of and tries the ones that are not part of any counter examples first. But the first candidate position that is not in any counter example is the one located so far off to the other side, that none of the counter examples are big enough to encompass it. Once this position has been chosen, no candidate positions going from that position will ever be a part of a counter example again, rendering the entire optimization useless. In effect, it places the second queen one spot further away from the corner than the largest counter example and the just consumes time for all other queen placements without providing any benefits.

It becomes clear that the optimization does not help the algorithm when looking that execution time for the tests where the maximum amount of tasks was utilized.

The execution time for N = 25 using zero additional tasks may seem wrong compared to the sizes before and after it. But it is in fact a good example of why the heuristic this optimization uses is not useful, since the solution for N = 25 is to simply place a queen at the first safe position and continue this almost until you reach the last row. It is clear that many of the early queen placements may be in some counter example, yet the fastest solution is to completely ignore this fact.

7 Conclusion

A parallelized version of the generic backtrack algorithm was successfully implemented for solving the N-Queen problem. The speedup achieved by the parallelization in the case of finding all solutions to the problem, was close to the number of cores used for executing the algorithm, which is what was expected based on an analysis of how large a portion of the program had to be serial in combination with Amdahl's Law. For finding only a single solution, such an analysis could not be performed, but the expected speedup was also near that of the number of cores present, yet the actual speedup gained greatly exceeded the expectation in many cases, sometimes upwards of 25 times larger than then number of physical cores used, provided the parameters for the parallelization were properly adjusted.

Some heuristics and optimizations were investigated and some implemented and tested, but only using the symmetry of the problem provided a reliable speedup when solving for all solutions. The heuristic for using Shortest Possible Counter Examples found in problems of a smaller size than the one that was attempted to be solved, slowed down the algorithm in most cases but also showed the importance of the early choices of queen placements.

The .NET Framework prove useful in the development of the parallel backtrack search, as their primitives allows a complete abstraction away from threads. The primitive Task was used where a thread would normally have been used if this was done on another platform, this primitive is control by a lot of heuristics behind the scenes, which help provide load balancing across the available cores and also makes the creation of a new Task much cheaper than a thread would have been. The Parallel.ForEach loop also yielded very nice results when used in conjunction with the algorithm for finding all solutions, once the problem size became large enough the speedup by the parallel loop was as close the number of physical cores as possible.

8 Appendix

1. Datagrid.cs

Contains partial and complete solutions, used as the primary data structure in the algorithm.

```
using System.Collections.Generic;

namespace NQPSolver
{
    public class DataGrid
    {
        private readonly List<Position> grid; // Positions of queen placements.
        public int ChangesMade { get; private set; }

        /// <summary>
        /// Used with the value based approach.
        /// </summary>
        /// <param name="n"></param>
        public DataGrid(int n)
        {
            Size = n;
            grid = new List<Position>(n);
        }

        /// <summary>
        /// Constructor used for cloning.
        /// </summary>
        /// <param name="data"></param>
        /// <param name="n"></param>
        /// <param name="changes"></param>
        private DataGrid(List<Position> data, int n, int changes)
        {
            this.grid = data;
            this.Size = n;
            this.ChangesMade = changes;
        }

        public int Size { get; private set; }

        /// <summary>
        /// Returns the current queen placements. Do not alter the elements.
        /// </summary>
        public List<Position> Grid
        {
            get
            {
                return new List<Position>(grid);
            }
        }

        public bool this[Position x]
        {
            get { return this[x.X, x.Y]; }
        }
```

```
        /// <summary>
        /// Lookup if a queen has been placed at the specifed position.
        /// </summary>
        /// <param name="x">X coordinate.</param>
        /// <param name="y">Y coordiante.</param>
        /// <returns>Returns true if a queen is at that position.</returns>
        public bool this[int x, int y]
        {
            get
            {
                if (ChangesMade <= y) return false;
                return grid[y].X == x ? true : false;
            }
        }

        public DataGrid Set(Position t)
        {
            return Set(t.X,t.Y);
        }

        /// <summary>
        /// Returns a new datagrid with a queen placed on x,y.
        /// </summary>
        /// <param name="x">X coordinate.</param>
        /// <param name="y">Y coordinate.</param>
        /// <returns>The new grid with the new queen placement.</returns>
        public DataGrid Set(int x, int y)
        {
            var clone = Clone();
            clone.grid.Add(new Position(x,y));
            clone.ChangesMade++;
            return clone;
        }

        /// <summary>
        /// Performs a shallow clone of this datagrid.
        /// </summary>
        /// <returns>A shallow clone of this datagrid.</returns>
        public DataGrid Clone()
        {
            return new DataGrid(new List<Position>(grid), Size, ChangesMade);
        }

        /// <summary>
        /// Reflects the specified datagrid around the Y-Axis.
        /// </summary>
        /// <param name="r">Grid to be mirrored.</param>
        /// <returns>A Y-Reflection of the datagrid.</returns>
        public static DataGrid YReflection(DataGrid r)
        {
            var rgrid = new List<Position>();

            for (int y = 0; y < r.Size; y++)
            {
                for (int x = 0; x < r.Size; x++)
                {
                    if (r[x,y])
                    {
                        int mirrorX = (r.Size - 1) - x;
                        rgrid.Add(new Position(mirrorX, y));
```

```
                        break;
                    }
                }
            }

            return new DataGrid(rgrid,r.Size,r.ChangesMade);
        }
    }
}
```

2. Solver.cs

Has the implementations of the algorithms used for solving the N-Queen problem, both for all solutions and a single, also has the algorithm for gathering SPCEs for use in the Counter Example Set.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace NQPSolver
{
    public class Solver
    {
        private readonly DataGrid dataGrid;
        private readonly object resultsLock = new object();
        private bool findSingle;
        private int taskCounter = 0; // Keeps track of current number of tasks.
        private volatile bool isDone;
        private CounterExampleSet counterExamples; // Stores SPCEs.

        public List<DataGrid> Results;
        public int MaxDepth { get; set; }
        public int MaxThreadCount { get; set; }
        public bool ShouldOptimize { get; set; }
        public int MinDepth { get; set; }

        public Solver(int n)
        {
            this.dataGrid = new DataGrid(n);
            Results = new List<DataGrid>();
            MinDepth = n/10;
        }

        public void Solve()
        {
            Solve(true);
        }

        public void Solve(object boolean)
        {
            bool f = (bool)boolean;
            Solve(f);
```

```
        }

        public void Solve(bool findSingleSolution)
        {
            findSingle = findSingleSolution;

            if (findSingle)
            {
                StartSolveSingle();
            }
            else
            {
                StartSolveAll();
            }
        }

        // Solve for all solutions.
        private void StartSolveAll()
        {
            ThreadPool.SetMaxThreads(8, 1);
            //Try to solve for each of the candidates.
            List<Position> candidateList = new List<Position>();
            if (ShouldOptimize)
            {
                candidateList = GetStartCandidates(dataGrid);
            }
            else
            {
                for (int i = 0; i < dataGrid.Size; i++)
                {
                    candidateList.Add(new Position(i,0));
                }
            }

            if (MaxThreadCount != 0)
            {
                Parallel.ForEach(candidateList, s => SolveAll(dataGrid, s));
            }
            else
            {
                foreach (var s in candidateList)
                {
                    SolveAll(dataGrid,s);
                }
            }

            if (ShouldOptimize)
            {
                var mirrored = new List<DataGrid>();
                foreach (var result in Results)
                {
                    for (int i = 0; i < result.Size / 2; i++)
                    {
                        if (result[i, 0])
                        {
                            mirrored.Add(DataGrid.YReflection(result));
                            break;
                        }
                    }
                }
```

```
            Results.AddRange(mirrored);
        }
    }

    private void SolveAll(DataGrid p, IEnumerable<Position> cs)
    {
        foreach (var tuple in cs)
        {
            SolveAll(p, tuple);
        }
    }

    private void SolveAll(DataGrid p, Position c)
    {
        var newP = p.Set(c);

        if (Accpeted(newP))
        {
            lock (resultsLock)
            {
                Results.Add(newP);
            }
            return;
        }

        var nexts = GetRow(newP, GetFirst(c)).FindAll(c1 => IsSafe(newP, c1));
        if (nexts.Count > 0)
        {
            SolveAll(newP.Clone(), nexts);
        }
    }

    // Solve for one solution
    private void StartSolveSingle()
    {
        //Start candidates. Only check one half of the board,
        //other half only contains mirrored versions of the first half's solutions.

        var candidateList = GetStartCandidates(dataGrid);
        counterExamples = new CounterExampleSet();

        if (ShouldOptimize)
        {
            Task csFinder = new Task(StartFindCounterExamples);
            csFinder.Start();
            csFinder.Wait();
        }

        foreach (var tuple in candidateList)
        {
            SolveSingle(dataGrid, tuple);
            if (isDone) break;
        }
    }

    /// <summary>
    /// Method used for parallelization.
    /// </summary>
    /// <param name="gridAndPosition"></param>
```

```
        private void SolveSingle(object gridAndPosition)
        {
            var parameters = (PSolverParameter)gridAndPosition;
            SolveSingle(parameters.DataGrid, parameters.Positions);
            Interlocked.Decrement(ref taskCounter);
        }

        private void SolveSingle(DataGrid p, IEnumerable<Position> cs)
        {
            foreach (var tuple in cs)
            {
                if (isDone) break;
                SolveSingle(p, tuple);
            }
        }

        /// <summary>
        /// The backtrack search, using Tasks for parallelization and terminates when a
single solution is found.
        /// </summary>
        /// <param name="p"></param>
        /// <param name="c"></param>
        private void SolveSingle(DataGrid p, Position c)
        {
            var newP = p.Set(c);

            if (Accpeted(newP))
            {
                isDone = true;
                lock (resultsLock)
                {
                    Results.Add(newP);
                }
            }

            if (isDone) return;

            var nexts = GetRow(p, GetFirst(c)).FindAll(c1 => IsSafe(newP, c1));

            if (ShouldOptimize)
            {
                nexts = counterExamples.SortList(nexts, newP);
            }

            if (taskCounter < MaxThreadCount && newP.ChangesMade > MinDepth
&&newP.ChangesMade < MaxDepth && nexts.Count > 1) //Nexts must contain more than one
element, else there is no reason to start a new thread.
            {
                List<Position> halfNexts = null;
                if (ShouldOptimize)
                {
                    halfNexts = FairListSplit(nexts);
                }
                else
                {
                    halfNexts = nexts.GetRange(0, nexts.Count/2);
                    nexts.RemoveRange(0,nexts.Count/2);
                }
                Interlocked.Increment(ref taskCounter);
                Task.Factory.StartNew(SolveSingle, new PSolverParameter(newP.Clone(),
```

```
halfNexts));
            }

            if (nexts.Count > 0)
            {
                SolveSingle(newP.Clone(), nexts);
            }
        }

        // Find counter examples in smaller problems.
        private void StartFindCounterExamples()
        {
            //var p = new DataGrid(4);
            //var cand = GetStartCandidates(p);
            //FindCounterExamples(p, cand);

            //int stop = 0;
            counterExamples = new CounterExampleSet();
            List<Task> tasks = new List<Task>();

            for (int i = 4; i < 12; i++)
            {
                var p = new DataGrid(i);
                var candidates = GetStartCandidates(p);
                Task task = new Task(FindCounterExamples, new PSolverParameter(p,
candidates));
                task.Start();
                tasks.Add(task);
            }

            foreach (var task in tasks)
            {
                task.Wait();
            }
        }

        private void FindCounterExamples(object gridAndPositions)
        {
            var parameters = (PSolverParameter) gridAndPositions;
            FindCounterExamples(parameters.DataGrid,parameters.Positions);
        }

        /// <summary>
        /// Adds SPCEs to the dictionary of SPCEs when found.
        /// </summary>
        /// <param name="p"></param>
        /// <param name="cs"></param>
        /// <returns></returns>
        private bool FindCounterExamples(DataGrid p, IEnumerable<Position> cs)
        {
            bool foundSolution = false;
            var possibleCounters = new List<DataGrid>();
            foreach (var tuple in cs)
            {
                if (FindCounterExamples(p, tuple))
                {
                    foundSolution = true;
                }
                else
                {
```

```
                    possibleCounters.Add(p.Set(tuple));
                }
            }

            if (!foundSolution && p.ChangesMade == 0)
            {
                //counterExamples.Add(p); // Everything is a counter example, p has no
solutions.
            }
            else if (foundSolution && possibleCounters.Count > 0)
            {
                counterExamples.Add(possibleCounters);
            }

            return foundSolution;
        }

        /// <summary>
        /// Part of the backtrack search for SPCEs.
        /// </summary>
        /// <param name="p"></param>
        /// <param name="c"></param>
        /// <returns></returns>
        private bool FindCounterExamples(DataGrid p, Position c)
        {
            var newP = p.Set(c);

            if (Accpeted(newP))
            {
                return true;
            }

            var nexts = GetRow(newP, GetFirst(c)).FindAll(c1 => IsSafe(newP, c1));

            bool foundSolution = false;
            if (nexts.Count > 0)
            {
                if (FindCounterExamples(newP.Clone(), nexts))
                    foundSolution = true;
            }

            return foundSolution;
        }

        // Misc functions.
        /// <summary>
        /// Used for splitting the sorted list in two pieces, ensuring both pieces have
viable positions.
        /// </summary>
        /// <param name="listToSplit"></param>
        /// <returns></returns>
        private static List<Position> FairListSplit(List<Position> listToSplit)
        {
            var firstHalf = new List<Position>();
            for (int i = 0; i < listToSplit.Count; i += 2)
            {
                firstHalf.Add(listToSplit[i]);
            }

            listToSplit.RemoveAll(firstHalf.Contains);
```

```
            return firstHalf;
        }

        /// <summary>
        /// Returns half of the positions of row zero in datagrid P.
        /// </summary>
        /// <param name="p"></param>
        /// <returns></returns>
        private static List<Position> GetStartCandidates(DataGrid p)
        {
            var candidateList = new List<Position>();
            int smartCandidates = (p.Size + 1) / 2;
            for (int i = 0; i < smartCandidates; i++)
            {
                candidateList.Add(new Position(i, 0));
            }

            return candidateList;
        }

        /// <summary>
        /// Returns true if N queens have been placed.
        /// </summary>
        /// <param name="newP"></param>
        /// <returns></returns>
        private static bool Accpeted(DataGrid newP)
        {
            return newP.ChangesMade == newP.Size ? true : false;
        }

        private static Position GetFirst(Position c)
        {
            return new Position(0,c.Y+1);
        }

        private static List<Position> GetRow(DataGrid p, Position c)
        {
            var cs = new List<Position>();

            int size = p.Size;
            for (int i = 0; i < size; i++)
            {
                cs.Add(new Position(i, c.Y));
            }

            return cs;
        }

        private static bool IsSafe(DataGrid p, Position c)
        {
            int size = p.Size;
            //Check vertical ONLY, horizontal is never an issue.
            for (int i = 0; i < size; i++)
            {
                if (p[c.X, i]) return false;
            }

            bool diags = IsDiagonalsSafe(p,c);
            if (!diags) return false;
```

```
            return true;
        }

        private static bool IsDiagonalsSafe(DataGrid p, Position c)
        {
            int size = p.Size;
            int max = size - 1;
            var start1 = new Tuple<int, int>(0,0);
            var start2 = new Tuple<int, int>(max, 0);
            int item1 = c.X;
            int item2 = c.Y;
            int posLength = size - 1, negLength = size - 1;

            //Below the Positive diagonal
            if (item2 - item1 < 0)
            {
                start1 = new Tuple<int, int>(item1 - item2, 0);
                posLength = size - start1.Item1 - 1;
            }
            //Above the Positive diagonal
            else if (item1 - item2 < 0)
            {
                start1 = new Tuple<int, int>(0, item2 - item1);
                posLength = size - start1.Item2 - 1;
            }

            for (int i = 0; i < posLength; i++)
            {
                if (p[start1.Item1 + i, start1.Item2 + i]) return false;
            }

            //Below the Negativ diagonal
            if (item1 + item2 < max)
            {
                start2 = new Tuple<int, int>(item1 + item2, 0);
                negLength = start2.Item1;
            }
            //Above the Negativ diagonal
            if (item1 + item2 > max)
            {
                start2 = new Tuple<int, int>(size - 1, item2 - (max - item1));
                negLength = size - start2.Item2 - 1;
            }

            for (int i = 0; i < negLength; i++)
            {
                if (p[start2.Item1 - i, start2.Item2 + i]) return false;
            }

            return true;
        }
    }
}
```

3. CounterExampleSet.cs

Stores and retrieve previously found counter examples. Can also sort a list of positions based on the number of counter examples each position is involved in.

```
using System.Collections.Generic;
using System.Linq;

namespace NQPSolver
{
    internal class CounterExampleSet
    {
        private readonly Dictionary<Position,ListDictPair> lookup = new
Dictionary<Position, ListDictPair>();
        private readonly object lookupLock = new object();

        public int MaxDepth { get; private set; }

        /// <summary>
        /// Adds the specifed counter examples to the dictionary. This is threadsafe.
        /// </summary>
        /// <param name="counterExamples">A grid containing a counter example.</param>
        public void Add(IEnumerable<DataGrid> counterExamples)
        {
            foreach (var dataGrid in counterExamples)
            {
                Add(dataGrid);
            }
        }

        /// <summary>
        /// Adds the specifed counter example to the dictionary. This is threadsafe.
        /// </summary>
        /// <param name="counterExample">A grid containing a counter example.</param>
        public void Add(DataGrid counterExample)
        {
            lock (lookupLock)
            {
                AddCounterExample(counterExample);
            }
        }

        /// <summary>
        /// Adds the specifed counter example to the dictionary.
        /// </summary>
        /// <param name="grid">A grid containing a counter example.</param>
        private void AddCounterExample(DataGrid grid)
        {
            var listpair = GetListDictPairs(grid.Grid);
            Fill(grid,listpair);
        }

        /// <summary>
        /// Adds a counter example to a listDictPair.
        /// </summary>
        /// <param name="counter">The CE to be added.</param>
```

```
        /// <param name="listDictPair">The lDP to have a CE added to.</param>
        private static void Fill(DataGrid counter, ListDictPair listDictPair)
        {
            listDictPair.CounterExamples.Add(counter);
        }

        private ListDictPair GetListDictPairs(List<Position> positions)
        {
            return GetListDictPairs(positions, lookup);
        }

        /// <summary>
        /// Returns the listDictPair found using the specified path, adds the listDictPair
if it did not exist.
        /// </summary>
        /// <param name="positions">The path used for lookup in the dictionary.</param>
        /// <param name="dictionary">Dictionary containing counter examples.</param>
        /// <param name="n">Recursion counter.</param>
        /// <returns></returns>
        private ListDictPair GetListDictPairs(List<Position> positions,
Dictionary<Position,ListDictPair> dictionary, int n = 0)
        {
            var newPair = new ListDictPair(new List<DataGrid>(), new Dictionary<Position,
ListDictPair>());

            if (n == positions.Count - 1)
            {
                if (MaxDepth < n) MaxDepth = n;
                if (dictionary.ContainsKey(positions[n]))
                {
                    return dictionary[positions[n]];
                }

                dictionary.Add(positions[n], newPair);
                return newPair;
            }

            if (!dictionary.ContainsKey(positions[n])) dictionary.Add(positions[n],
newPair);

            return GetListDictPairs(positions, dictionary[positions[n]].Dictionary, n + 1);
        }

        /// <summary>
        /// Sorts the list of positions based on how many counter examples such a position
would be part of.
        /// </summary>
        /// <param name="listToSort">The list to be sorted.</param>
        /// <param name="currentGrid">The current grid.</param>
        /// <returns>The list sorted based on the number counter examples each position
would be part of.</returns>
        public List<Position> SortList(List<Position> listToSort, DataGrid currentGrid)
        {
            if (MaxDepth < currentGrid.ChangesMade) return listToSort; //No counter
examples can match.

            //Order listToSort based on counterExamples.
            IEnumerable<DataGrid> matchingCounters = GetCounterExamples(currentGrid);
            if (matchingCounters == null || matchingCounters.Count() == 0) return
listToSort;
```

```
            var weights = AddWeights(listToSort);

            foreach (var matchingCounter in matchingCounters)
            {
                foreach (var weightedPosition in weights)
                {
                    if (weightedPosition.Position.HorizontalDistance(currentGrid.Grid[0]) <
matchingCounter.Size)
                    {
                        weightedPosition.Weight++;
                    }
                }
            }

            weights.Sort();

            return StripWeights(weights);
        }

        /// <summary>
        /// Takes a list of positions and returns a list with the same positions but with
each one having a weight.
        /// </summary>
        /// <param name="positions">The list of which to add weights.</param>
        /// <returns>The list with added weights.</returns>
        private List<WeightedPosition> AddWeights(List<Position> positions)
        {
            return positions.Select(p => new WeightedPosition(p)).ToList();
        }

        /// <summary>
        /// Removes weigths from a list with positions and weights.
        /// </summary>
        /// <param name="weightedPositions">List to have weights removed from.</param>
        /// <returns>List without the weighs.</returns>
        private List<Position> StripWeights(List<WeightedPosition> weightedPositions)
        {
            return weightedPositions.Select(position => position.Position).ToList();
        }

        private IEnumerable<DataGrid> GetCounterExamples(DataGrid currentGrid)
        {
            return GetCounterExamples(currentGrid, lookup);
        }

        /// <summary>
        /// Gives the counter examples the current path is involved in.
        /// </summary>
        /// <param name="currentGrid">The current grid containing a path.</param>
        /// <param name="dictionary">The dictionary containing counter examples.</param>
        /// <param name="n">Recursion counter.</param>
        /// <returns>Returns counter examples for the current grid.</returns>
        private static IEnumerable<DataGrid> GetCounterExamples(DataGrid currentGrid,
Dictionary<Position,ListDictPair> dictionary, int n = 0)
        {
            Position index = currentGrid.Grid[n];
            if (n == currentGrid.Grid.Count - 1)
            {
                return dictionary.ContainsKey(index) ? dictionary[index].CounterExamples :
```

```
null;
            }

            return dictionary.ContainsKey(index) ? GetCounterExamples(currentGrid,
dictionary, n + 1) : null;
        }
    }
}
```

4. WeightedPosition.cs

Used in the Counter Example Set for keep track of the weights assigned to each position. The weighted position is not used in the algorithm itself, as the weight attribute is only used during the sorting of the position list.

```
using System;

namespace NQPSolver
{
    internal class WeightedPosition : IComparable<WeightedPosition>
    {
        public Position Position { get; private set; }
        public int Weight { get; set; }

        /// <summary>
        /// A position with a weight, implements IComparable to allow sorting.
        /// </summary>
        /// <param name="p"></param>
        /// <param name="weight"></param>
        public WeightedPosition(Position p, int weight = 0)
        {
            Position = p;
            Weight = weight;
        }

        public int CompareTo(WeightedPosition other)
        {
            if (this.Weight < other.Weight)
            {
                return -1;
            }
            else if (other.Weight < this.Weight)
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
    }
}
```

5. SPSolverParameter.cs

Used by the reference based solver to pass parameters to Tasks. Currently unused.

```
using System;
using System.Collections.Generic;

namespace NQPSolver
{
    internal struct SPSolverParameter
    {
        public SmartDataGrid SmartDataGrid;
        public List<Tuple<int, int>> Positions;

        /// <summary>
        /// Used with the reference based approach. Discontinued.
        /// </summary>
        /// <param name="p"></param>
        /// <param name="cs"></param>
        public SPSolverParameter(SmartDataGrid p, List<Tuple<int, int>> cs)
        {
            this.SmartDataGrid = p;
            this.Positions = cs;
        }
    }
}
```

6. SmartSolver.cs

Solves the problem using mostly references to data grid, instead of copying them every time.

Currently unused and not updated recently.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace NQPSolver
{
    public class SmartSolver
    {
        private readonly SmartDataGrid smartDataGrid;
        private int taskCounter = 0;
        private volatile bool isDone;

        /// <summary>
```

```
        /// Used with the reference based approach. Discontinued.
        /// </summary>
        /// <param name="n"></param>
        public SmartSolver(int n)
        {
            this.smartDataGrid = new SmartDataGrid(n);
        }

        public int Depth { get; set; }
        public int MaxThreadCount { get; set; }

        public void SSolve()
        {
            //Start candidates. Only check one half of the board,
            //other half only contains mirrored versions of the first half's solutions.
            var candidateList = new List<Tuple<int, int>>();
            int smartCandidates = (smartDataGrid.Size / 2) + 1;
            for (int i = 0; i < smartCandidates; i++)
            {
                candidateList.Add(new Tuple<int, int>(i, 0));
            }
            //ThreadPool.SetMaxThreads(8, 8);
            //Try to solve for each of the candidates.
            //Parallel.ForEach(candidateList, (s) => Solve(dataGrid, s));)
            foreach (var tuple in candidateList)
            {
                SSolve(smartDataGrid, tuple);
                if (isDone) break;
            }
        }

        private void SParallelSolve(object gridAndPosition)
        {
            var parameters = (SPSolverParameter)gridAndPosition;
            SSolve(parameters.SmartDataGrid, parameters.Positions);
            Interlocked.Decrement(ref taskCounter);
        }

        private void SSolve(SmartDataGrid p, IEnumerable<Tuple<int, int>> cs)
        {
            foreach (var tuple in cs)
            {
                if (isDone) break;
                SSolve(p, tuple);
            }
        }

        public SmartDataGrid Result;
        private readonly object resultLock = new object();

        private void SSolve(SmartDataGrid p, Tuple<int, int> c)
        {

            p.SetToTrue(c);

            if (Accpeted(p))
            {
                isDone = true;
                lock (resultLock)
                {
```

```
                    Result = p;
                }
            }

            if (isDone) return;

            var nexts = GetRow(p, GetFirst(c)).FindAll(c1 => IsSafe(p, c1));
            if (taskCounter < MaxThreadCount && p.ChangesMade < Depth && nexts.Count > 1)
//Nexts must contain more than one element, else there is no reason to start a new thread.
            {
                var halfNexts = nexts.GetRange(0, nexts.Count / 2);
                nexts.RemoveRange(0, nexts.Count / 2);
                Interlocked.Increment(ref taskCounter);
                Task.Factory.StartNew(SParallelSolve, new SPSolverParameter(p.Clone(),
halfNexts));
            }

            if (nexts.Count > 0)
            {
                SSolve(p, nexts);
            }

            p.UndoLatestChange();
        }

        private bool Accpeted(SmartDataGrid newP)
        {
            return newP.ChangesMade == newP.Size ? true : false;
        }

        private Tuple<int,int> GetFirst(Tuple<int, int> c)
        {
            return new Tuple<int, int>(0,c.Item2+1);
        }

        private List<Tuple<int, int>> GetRow(SmartDataGrid p, Tuple<int, int> c)
        {
            var cs = new List<Tuple<int, int>>();

            int size = p.Size;
            for (int i = 0; i < size; i++)
            {
                cs.Add(new Tuple<int, int>(i, c.Item2));
            }

            return cs;
        }

        private bool IsSafe(SmartDataGrid p, Tuple<int, int> c)
        {
            int size = p.Size;
            //Check vertical and horizontal
            for (int i = 0; i < size; i++)
            {
                if (p[c.Item1, i]) return false;
                if (p[i, c.Item2]) return false;
            }

            bool diags = IsDiagonalsSafe(p, c);
            if (!diags) return false;
```

```
            return true;
        }

        private bool IsDiagonalsSafe(SmartDataGrid p, Tuple<int, int> c)
        {
            int size = p.Size;
            int max = size - 1;
            var start1 = new Tuple<int, int>(0, 0);
            var start2 = new Tuple<int, int>(max, 0);
            int posLength = size - 1, negLength = size - 1;

            //Below the Positive diagonal
            if (c.Item2 - c.Item1 < 0)
            {
                start1 = new Tuple<int, int>(c.Item1 - c.Item2, 0);
                posLength = size - start1.Item1 - 1;
            }
            //Above the Positive diagonal
            else if (c.Item1 - c.Item2 < 0)
            {
                start1 = new Tuple<int, int>(0, c.Item2 - c.Item1);
                posLength = size - start1.Item2 - 1;
            }

            for (int i = 0; i < posLength; i++)
            {
                if (p[start1.Item1 + i, start1.Item2 + i]) return false;
            }

            //Below the Negativ diagonal
            if (c.Item1 + c.Item2 < max)
            {
                start2 = new Tuple<int, int>(c.Item1 + c.Item2, 0);
                negLength = start2.Item1;
            }
            //Above the Negativ diagonal
            if (c.Item1 + c.Item2 > max)
            {
                start2 = new Tuple<int, int>(size - 1, c.Item2 - (max - c.Item1));
                negLength = size - start2.Item2 - 1;
            }

            for (int i = 0; i < negLength; i++)
            {
                if (p[start2.Item1 - i, start2.Item2 + i]) return false;
            }

            return true;
        }
    }
}
```

7. SmartDataGrid.cs

The data grid used in the reference based approach, has additional data to allow undoing of queen placements. Unused and outdated.

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace NQPSolver
{
    public class SmartDataGrid
    {
        private readonly BitArray grid; // Flattend 2D array.
        private readonly Stack<Tuple<int, int>> changeHistory;

        /// <summary>
        /// Used with the reference based approach. Discontinued.
        /// </summary>
        /// <param name="n"></param>
        public SmartDataGrid(int n)
        {
            Size = n;
            grid = new BitArray(n*n);
            changeHistory = new Stack<Tuple<int, int>>(n);
        }

        private SmartDataGrid(BitArray data, int n, Stack<Tuple<int,int>> changeHistory)
        {
            this.grid = data;
            this.Size = n;
            this.changeHistory = changeHistory;
        }

        public int ChangesMade { get { return changeHistory.Count; } }
        public int Size { get; private set; }

        public bool this[Tuple<int,int> c]
        {
            get { return this[c.Item1, c.Item2]; }
            private set { this[c.Item1, c.Item2] = value; }
        }

        public bool this[int x, int y]
        {
            get
            {
                return this.grid[x + (y * Size)] ? true : false;
            }
            private set { this.grid[x + (y * Size)] = value; }
        }

        /// <summary>
        /// Undo the lastest queen placement.
        /// </summary>
        public void UndoLatestChange()
        {
            var position = changeHistory.Pop();
            this[position] = false;
        }

        /// <summary>
        /// Place a queen at position T.
```

```
        /// </summary>
        /// <param name="t"></param>
        public void SetToTrue(Tuple<int,int> t)
        {
            SetToTrue(t.Item1, t.Item2);
        }

        public void SetToTrue(int x, int y)
        {
            this[x, y] = true;
            changeHistory.Push(new Tuple<int, int>(x,y));
        }

        /// <summary>
        /// Performs a shallow cloning of the SmartDataGrid.
        /// </summary>
        /// <returns>Returns a shallow clone.</returns>
        public SmartDataGrid Clone()
        {
            return new SmartDataGrid(new BitArray(this.grid), Size, new Stack<Tuple<int,
int>>(changeHistory));
        }
    }
}
```

8. SBacktrackFinishedEventArgs.cs

Used by the reference based solver to pass information to the interface. Unused.

```
using System;

namespace NQPSolver
{
    public class SBacktrackFinishedEventArgs : EventArgs
    {
        public SmartDataGrid SmartDataGrid { get; private set; }

        public SBacktrackFinishedEventArgs(SmartDataGrid smartDataGrid)
        {
            this.SmartDataGrid = smartDataGrid;
        }
    }
}
```

9. PSolverParameter.cs

Used by the value based solver to pass parameters to newly created Tasks.

```
using System.Collections.Generic;

namespace NQPSolver
{
    internal struct PSolverParameter
    {
        public DataGrid DataGrid;
        public List<Position> Positions;

        public PSolverParameter(DataGrid p, List<Position> cs)
        {
            this.DataGrid = p;
            this.Positions = cs;
        }
    }
}
```

10. Position.cs

Represents a position in an x-y coordinate system, supports value based equality and can generate a hashcode, both for use with the dictionary in Counter Example Set.

```
using System;

namespace NQPSolver
{
    public struct Position
    {
        public int X;
        public int Y;

        public Position(int x, int y)
        {
            X = x;
            Y = y;
        }

        /// <summary>
        /// Equality check by value. Required for use as key in a dictionary.
        /// </summary>
        /// <param name="obj">Object to check equality against.</param>
        /// <returns>Returns true if the X and Y coordinates are identical, otherwise
false.</returns>
        public override bool Equals(object obj)
        {
            if (ReferenceEquals(null, obj)) return false;
            if (obj.GetType() != typeof (Position)) return false;
            return Equals((Position) obj);
        }
```

```
        public bool Equals(Position other)
        {
            return other.X == X && other.Y == Y;
        }

        /// <summary>
        /// Get the horizontal distance between two positions.
        /// </summary>
        /// <param name="other"></param>
        /// <returns></returns>
        public int HorizontalDistance(Position other)
        {
            return Math.Abs(this.X - other.X);
        }

        /// <summary>
        /// Generates a hashcode for the Position object. Required for use as key in a
dictionary.
        /// </summary>
        /// <returns></returns>
        public override int GetHashCode()
        {
            unchecked
            {
                return X ^ Y;
            }
        }
    }
}
```

11. ListDictPair.cs

The base of the recursive data structure used in the Counter Example Set.

```
using System.Collections.Generic;

namespace NQPSolver
{
    internal struct ListDictPair
    {
        public List<DataGrid> CounterExamples;
        public Dictionary<Position, ListDictPair> Dictionary;

        public ListDictPair(List<DataGrid> counters, Dictionary<Position,ListDictPair>
dictionary)
        {
            CounterExamples = counters;
            Dictionary = dictionary;
        }
    }
}
```

12. BacktrackFinishedEventArgs.cs

Used for communication with the interface.

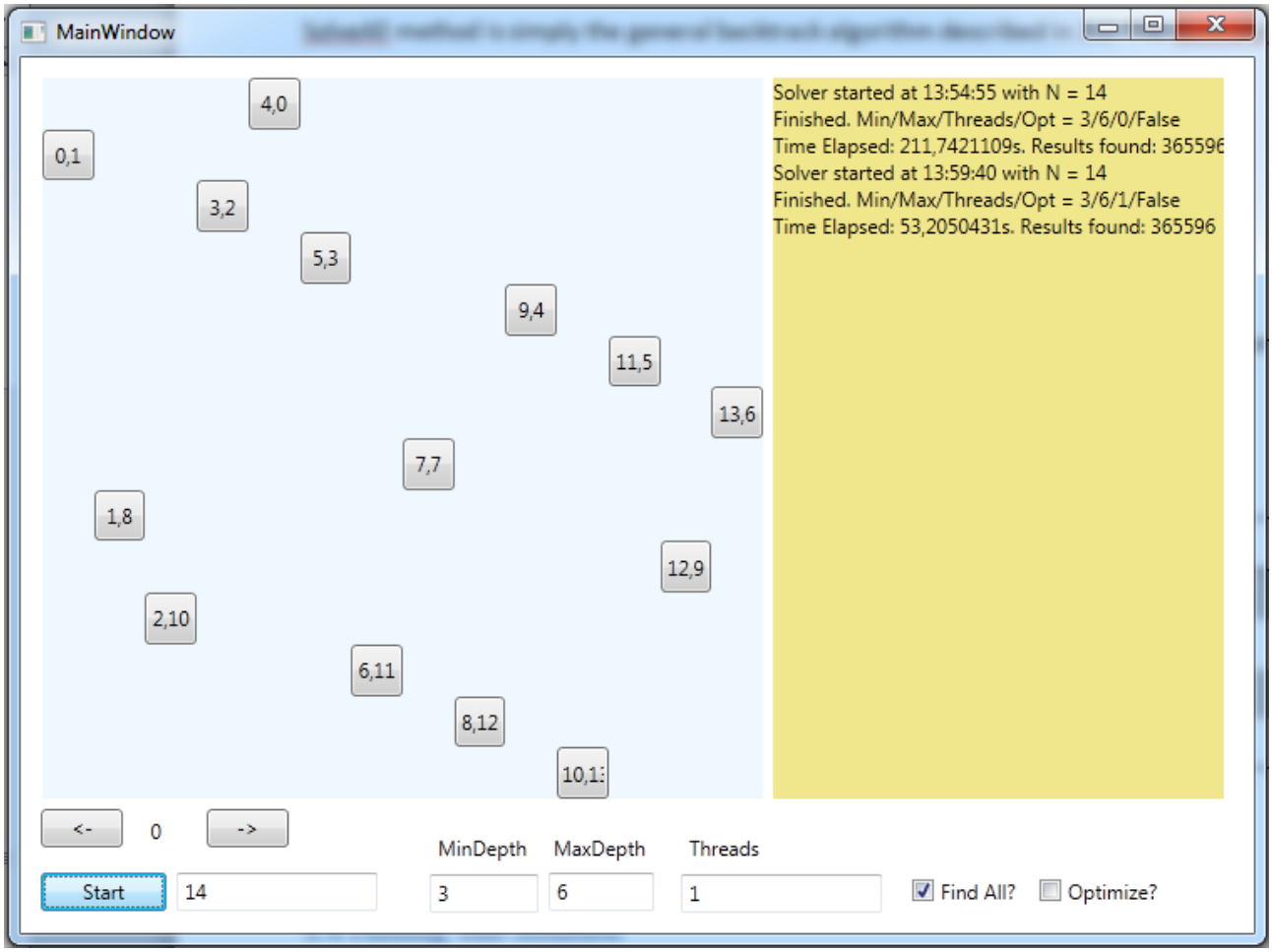
```
using System;

namespace NQPSolver
{
    public class BacktrackFinishedEventArgs : EventArgs
    {
        public DataGrid DataGrid { get; private set; }

        public BacktrackFinishedEventArgs(DataGrid dataGrid)
        {
            DataGrid = dataGrid;
        }
    }
}
```

13. MainWindow.xaml

Screenshot:



Code:

```
<Window x:Class="QueenVisualizer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="560" Width="752">
    <Grid>
        <Button Content="Start" Height="23" HorizontalAlignment="Left" Margin="12,486,0,0"
Name="StartButton" VerticalAlignment="Top" Width="75" Click="StartButton_Click" />
        <TextBox Height="23" HorizontalAlignment="Left" Margin="93,486,0,0" Name="ValueBox"
VerticalAlignment="Top" Width="120" />
        <Grid Name="ChessBoard" Height="430" HorizontalAlignment="Left" Margin="13,12,0,0"
VerticalAlignment="Top" Width="430" Background="AliceBlue" />
        <TextBlock Height="430" HorizontalAlignment="Left" Margin="449,12,0,0"
Name="ConsoleWindow" Text="" VerticalAlignment="Top" Width="269" Background="Khaki" />
        <Label Content="MinDepth" Height="28" HorizontalAlignment="Left"
Margin="244,458,0,0" Name="label1" VerticalAlignment="Top" />
        <Label Content="Threads" Height="28" HorizontalAlignment="Left"
```

```
Margin="394,458,0,0" Name="label2" VerticalAlignment="Top" />
        <TextBox Text="6" Height="23" HorizontalAlignment="Left" Margin="315,486,0,0"
Name="MaxDepthTextBox" VerticalAlignment="Top" Width="63" />
        <TextBox Text="0" Height="23" HorizontalAlignment="Left" Margin="394,487,0,0"
Name="ThreadsTextBox" VerticalAlignment="Top" Width="120" />
        <Button Content="" Height="23" HorizontalAlignment="Left" Margin="12,448,0,0"
Name="leftButton" VerticalAlignment="Top" Width="49" Click="leftButton_Click" />
        <Button Content="" Height="23" HorizontalAlignment="Left" Margin="111,448,0,0"
Name="rightButton" VerticalAlignment="Top" Width="49" Click="rightButton_Click" />
        <Label Content="0" Height="28" HorizontalAlignment="Right" Margin="0,448,647,0"
Name="resultLabel" VerticalAlignment="Top" />
        <CheckBox Content="Find All?" Height="16" HorizontalAlignment="Left"
Margin="532,489,0,0" Name="findAllCheckBox" VerticalAlignment="Top" />
        <CheckBox Content="Optimize?" Height="16" HorizontalAlignment="Left"
Margin="608,489,0,0" Name="shouldOptimizeCheckBox" VerticalAlignment="Top" />
        <Label Content="MaxDepth" Height="28" HorizontalAlignment="Left"
Margin="313,458,0,0" Name="label3" VerticalAlignment="Top" />
        <TextBox Height="23" HorizontalAlignment="Left" Margin="244,487,0,0"
Name="MinDepthTextBox" VerticalAlignment="Top" Width="65" Text="3" />
    </Grid>
</Window>
```

14. MainWindow.xaml.cs

This tool was meant purely for debugging during the design process, as it displays solutions in a readable fashion and allows tweaking parameters without modifying the code. The tool is not finished and suffers from very occasional crashes, but is still suited for performing tests.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using NQPSolver;
using DataGrid = NQPSolver.DataGrid;

namespace QueenVisualizer
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
```

```
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            leftButton.Content = "<-";
            rightButton.Content = "->";
            resultLabel.Content = currentResult;
        }



        private void StartButton_Click(object sender, RoutedEventArgs e)
        {

            var start = DateTime.Now;
            Solver solver = null;
            try
            {
                solver = new Solver(Int32.Parse(ValueBox.Text));

            } catch(Exception ee) {}

            if (solver == null) return;

            solver.MaxDepth = Int32.Parse(MaxDepthTextBox.Text);
            solver.MinDepth = Int32.Parse(MinDepthTextBox.Text);
            solver.MaxThreadCount = Int32.Parse(ThreadsTextBox.Text);
            solver.ShouldOptimize = shouldOptimizeCheckBox.IsChecked.Value;

            ConsoleWindow.Text += "Solver started at "+ start.ToLongTimeString() +" with N =
" + ValueBox.Text + "\n";
            Thread lol = new Thread(new ParameterizedThreadStart(solver.Solve));
            if (findAllCheckBox.IsChecked != null)
                if (!findAllCheckBox.IsChecked.Value)
                {
                    lol.Start(true);
                }
                else
                {
                    lol.Start(false);
                }
            lol.Join();
            var end = DateTime.Now;
            ConsoleWindow.Text += "Finished. Min/Max/Threads/Opt =
"+solver.MinDepth+"/"+solver.MaxDepth+"/"+solver.MaxThreadCount+"/"+shouldOptimizeCheckBox.I
sChecked.Value+" \n";
            ConsoleWindow.Text += "Time Elapsed: " + (end - start).TotalSeconds.ToString() +
"s. Results found: "+ solver.Results.Count +"\n";

            resultLabel.Content = 0;
            currentResult = 0;
            results = solver.Results;
            Draw(results[0]);
        }

        private int currentResult;
        private List<DataGrid> results = new List<DataGrid>();
```

```
        private void Draw(DataGrid dataGrid)
        {
            ChessBoard.RowDefinitions.Clear();
            ChessBoard.ColumnDefinitions.Clear();
            ChessBoard.Children.Clear();

            for (int i = 0; i < dataGrid.Size; i++)
            {
                ChessBoard.RowDefinitions.Add(new RowDefinition());
                ChessBoard.ColumnDefinitions.Add(new ColumnDefinition());
            }

            for (int x = 0; x < dataGrid.Size; x++)
            {
                for (int y = 0; y < dataGrid.Size; y++)
                {
                    if (dataGrid[x,y])
                    {
                        var button = new Button();
                        ChessBoard.Children.Add(button);
                        Grid.SetRow(button, y);
                        Grid.SetColumn(button, x);
                        button.Content = x + "," + y;
                    }
                }
            }
        }

        private void leftButton_Click(object sender, RoutedEventArgs e)
        {
            if (currentResult == 0)
            {

            } else
            {
                currentResult -= 1;
                resultLabel.Content = currentResult;
                Draw(results[currentResult]);
            }
        }

        private void rightButton_Click(object sender, RoutedEventArgs e)
        {
            if (currentResult == results.Count - 1)
            {

            } else
            {
                currentResult += 1;
                resultLabel.Content = currentResult;
                Draw(results[currentResult]);
            }
        }
    }
}
```

9 References

- http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf
- http://msdn.microsoft.com/en-us/library/dd460693.aspx
- Modern Operation Systems. Third Edition. By Andrew S. Tanenbaum.
- Comparison of Heuristic Algorithms for the N-Queen Problem. By Ivica Martinjak and Martin Golub. http://www.zemris.fer.hr/~golub/clanci/iti2007.pdf