Parallel Tree Search for the n-Queens Problem

Umeå University

Parallel Programming (5DV152)

Salome Müller, mcs21smr

March 16, 2022

Introduction

In this work we develop and test a parallel algorithm to count the number of solutions for the n-queens problem. This problem is about setting n queens on a $n \times n$ chess-board, such that no queen is in the same row, column or diagonal as any other queen. Such solutions can be found with a tree search. The nodes of this tree are configurations of the board the root node is the empty board. The children of a node always contain the same configuration of the board as the node itself, but with a new queen in the first empty row. In order to decrease the branching factor, sub-trees are pruned off as soon as one queen on the board violates any of the given constraints, i.e., is in the same row, column or diagonal as another queen.

Section 1 shows the algorithms that were used. We first talk about the sequential approach, before we go into detail on how to construct the parallel algorithm. Section 2 introduces the experiments with which we tested the algorithms. The results are shown in Section 4, and discussed in Section 5.

1 Algorithms

In this section we show the serial and the parallel algorithms that are used to compute the number of solutions to the n-queens problem. The two algorithms differ from each other but have some concepts in common.

Both store the representation of a board with $i < n$ set queens as a one dimensional array of length i. At index row, $row < i$, the array holds the index of the column col, at which the queen is positioned. We call this array q_pos.

Both algorithms use some form of backtracking to traverse the nodes of the search tree. This means, we add another queen to a valid configuration. If this queen does not violate any of the constraints either, we continue by adding queens, until n queens are placed on the board. If n queens are placed, and the configuration is valid, we increase the number of found solutions by one. If the queen violates a constraint, we remove it again and set it in the next possible position. Whenever a queen violates a constraint or completes a solution (being the n-th queen set), we replace that queen to another column in the same row. If no other column is available, we remove the queen and re-position the queen in the preceding row.

In order to inspect whether a newly set queen violates the constraints, both algorithms call the function shown in Algorithm 1. This function checks whether the last set queen is in the same column or diagonal as any queen on a higher row. Since queens are set row-wise, they cannot be in the same row as another queen. Further, since the method is called for every queen that is added, only the newly set queen must be checked. The other queens are assumed to not violate any constraints with each other.

Algorithm 1 Algorithm to validate a configuration

```
1: function isValid(q_pos)
2:     row ← q_pos.size - 1
3:     col ← q_pos[row]
4:     for r ∈ {0, 1, ..., row} do
5:         if q_pos[r] = col or
            r - q_pos[r] = row - col or
            r + q_pos[r] = row + col then
6:             return false
7:     return true
```

1.1 Serial

The serial algorithm is a recursive form of a depth-first tree search with pruning. Algorithm 2 shows the pseudo-code for this algorithm.

Algorithm 2 Serial algorithm for n-queens tree-search

```
1: function Queens(n, row, q_pos)
2:     if row >= n then
3:         return 1
4:     solutions ← 0
5:     for col ∈ {0, 1, ...,n−1} do
6:         q_pos_new ← q_pos.apped(col)
7:         if isValid(q_pos_new) then
8:             solutions ← solutions + Queens(n, row + 1, q_pos_new)
9:     return solutions
```

The procedure gets the number of queens n, the index of the next queen that should be set, *row*, and the current configuration of the board, *q_pos*.

At each recursive call, the algorithm first checks if the board is full (line 2). If so, it returns 1 since one valid solution was found. Otherwise the next queen is set to all possible values (line 6). If the resulting configuration is still a valid solution, i.e., the new queen does not violate any of the constraints, the process is recursively repeated and the number of solutions that can be reached from this configuration is added to the total number of solutions.

1.2 Parallel

The main challenge when transforming this algorithm into a parallel one is to ensure that all threads will get a similar workload. One is tempted to produce a number of configurations equal to the number of threads, and then split them between the threads, such that they can work in parallel on the resulting sub-trees. This approach could lead to some threads falling idle long before the last thread finishes, as there is no way to know in advance how many nodes of

the subtree can be pruned away. Thus, before parallelizing the algorithm we transform it a little by using a queue instead of recursive calls.

The queue is initialized by pushing all n configurations with only one queen. The algorithm shown in Algorithm 3 then pops one *q_pos* from the queue, creates all its successors, and pushes the valid ones to the queue again.

Algorithm 3 Parallel algorithm for n-queens tree-search

```
1:  function QueenParallel(n)
2:      queue ← initializeQueue()
3:      for col ∈ {0, 1, ...,n−1} do
4:          q_pos[0] ← col
5:          queue.push(q_pos)
6:
7:      solutions ← 0
8:      while queue.isNotEmpty() do
9:          q_pos ← queue.pop()
10:         for col ∈ {0, 1, ...,n−1} do
11:             q_pos_new ← q_pos.apped(col)
12:             if isValid(q_pos_new) then
13:                 if q_pos_new.isComplete() then
14:                     solutions ← solutions + 1
15:                 else
16:                     queue.push(q_pos_new)
17:     return solutions
```

When using multiple threads, the queue is shared between them. Any operation of the queue, i.e., push or pop elements, is a critical section. Therefore, and since the queue is shared between the threads, parallelization could not bring any improvement compared to the serial algorithm. All threads would repeatedly and concurrently access the queue, which would result in a nearly sequential computation.

Further, depending on the order in which the nodes are retrieved from the queue and the size of the problem (n), this approach can easily lead to memory overflow. One solution to this would be to use a priority queue, where solutions with more queens are popped first. However, the expected running time to add elements to priority queues is higher than for normal queues. Regarding that operations on the queue are critical sections that can only be executed by one thread at a time, the time for single operations on them should be kept minimal.

To solve both the problem of overflowing memory and repeated access to the same critical section, we introduce an array of queues, called *queues*. This array has length $n-1$, and at every position i it holds a queue containing only configurations with i set queens. When a thread needs a new configuration to work on, it pops the thread from the non-empty queue with highest index, e.g., q. It then creates is successors and pushes them to the queue with index $q+1$. Every queue is protected with its own lock, thus different threads can access

different queues simultaneously.

The time to process one node this way is very fast, as only n successors are created, validated and pushed. Therefore, every thread produces a lot of new nodes very fast, and the memory still overflows eventually.

We therefore reduce the size of *queues* by half. Now, the queue at index i contains all configurations with $2 * i$ queens. When a thread gets a new configuration it must thus produce all valid successors of the node that have two additional queens. This forces the thread to be busy with one node for a little longer, while it reduces the number of stored configurations and accesses to the queues. It still retains the benefit, that the work is split in small enough pieces to prevent threads from being idle for long periods.

Because we want *queues* to be shared among the threads, we use OpenMP. Algorithm 4 shows the pseudo code of the final implementation.

Algorithm 4 Parallel algorithm for n-queens tree-search

```
1:  function QueenParallel(n)
2:      global queues[]
3:      for col ∈ {0, 1, ...,n−1} do
4:          queues[i] ← initializeQueue()
5:
6:      for col ∈ {0, 1, ...,n−1} do
7:          q_pos[0] ← col
8:          queue[0].push(q_pos)
9:
10:     solutions ← 0
11:     #pragma omp parallel reduction(+:solutions)
12:     {
13:     my_sol ← 0
14:     while noQueueEmpty(queues) do
15:         q_pos, i_queue ← queue.pop()
16:         if q_pos.size < n then
17:             my_sol ← my_sol + append1Final(q_pos)
18:         else if q_pos.size + 1< n then
19:             my_sol ← my_sol + append2Final(q_pos)
20:         else
21:             append2(q_pos, queues[i_queue+1])
22:     solutions ← solutions + my_sol
23:     }
24:     return solutions
```

On lines 1 to 3 we initialise the array of queues, which contains n/2 queues for n queens. On lines 5 to 7 we create all configurations with only one queen and add them to the first queue. After this, we can start the parallel computation. Starting from line 10, each thread pops configurations from the non-empty queue

with highest index (i_queue) while none of the queues is empty. For every configuration there are three possibilities: Either q_pos lacks exactly one or two queens to be a final configuration. If so we add those queens and add the number of solutions which are produced this way to the private variable my_sol (on lines 16 and 18, respectively). If more queens are still missing, we add two more, and push the resulting valid configurations to the next queue (line 20).

When all queues are empty, the number of solutions the thread found (my_sol) is added to the total number of solutions. This can be done by using the OpenMP built-in reduction, which is the only critical section apart from the push and pop operations on the queues.

In the next section, we show how we test Algorithms 2 and 4.

2 Experiments

We implement the algorithms of Section 1 in C, for details on the implementation refer to the GitHub repository.

All experiments are performed on Kebnekaise. For each configuration we execute the program 5 times and take the average running time of these runs.

We have two different types of experiments. For the first one we choose a fixed number of queens, and compare the run times when using different numbers of threads. In order to have interesting test results, we choose the smallest n, for which the serial algorithm has a running time of about 10 minutes. This is $n = 16$, for which the serial algorithm (Algorithm 2) has a running time of 618 seconds on average. The number of threads runs from 1 to 28, which is the number of cores on Kebnekaise's compute nodes.

For the second one, we set the number of threads and solve the problem for increasing numbers of queens. We choose 8 and 28 as the number of threads, as 8 turns out to be the number of threads with highest speedup in the first experiment. We range n from 10 to 17. For less than 10 queens the problem can be solved within an instant, and 17 queens is the largest problem that can be solved within a reasonable amount of time with 8 threads.

3 Results

Increasing Number of Threads Table 1 shows the running times of the parallel algorithm with different numbers of threads on the 16-queens problem. The following formulas were used to compute the speedup and the efficiency:

$$S = \frac{T_{serial}}{T_{parallel}}, \qquad E = \frac{S}{NumberOfThreads}.$$

Both are computed based on the time the serial algorithm takes on the 16-queens problem, i.e., 618 seconds.

Threads	1	2	4	8	12	16	20	24	28
Time	1431	908	497	286	562	593	764	819	851
Speedup	0.43	0.68	1.24	2.16	1.10	1.04	0.81	0.76	0.73
Efficiency	0.43	0.34	0.31	0.27	0.09	0.07	0.04	0.03	0.03

Table 1: Running times in seconds, speedup and efficiency for different numbers of threads, solving the 16-queens problem

Increasing Number of Queens Table 2 shows the running times, speed up and efficiency for the serial algorithm and the parallel algorithm with 8 threads and 28 threads. The speedup and efficiency computed for the parallel algorithm are based on the serial running times displayed in the table.

Queens	10	11	12	13	14	15	16	17
Serial	0.01	0.07	0.29	1.8	11	77	618	4721
8 Threads	0.01	0.04	0.18	1.1	6.2	54	286	2004
Speedup	1.43	1.97	1.61	1.65	1.80	1.44	2.16	2.03
Efficiency	0.18	0.25	0.20	0.21	0.23	0.18	0.27	0.25
28 Threads	0.03	0.11	0.53	3.2	17	147	851	4911
Speedup	0.40	0.70	0.55	0.57	0.63	0.53	0.73	0.96
Efficiency	0.01	0.02	0.02	0.02	0.02	0.02	0.03	0.03

Table 2: Running times in seconds for different numbers of Queens using the serial algorithms, the parallel one with 8 and 28 threads, as well as the speedup and efficiency for the parallel results.

4 Discussion

Increasing Number of Threads When fixing the number of queens to 16, and increasing the number of threads, we see that the best running time is achieved with 8 threads, with a speedup of 2.16 and an efficiency of 0.27. The best efficiency is 0.43, achieved with only one thread. Surprisingly, more threads do not lead to faster running times with our algorithm.

Increasing Number of Queens In order to verify whether this holds only for the 16-queens problem, or for any number of queens, we further analyse the running times of 8 and 28 threads on instances of different sizes. The result is, that 8 threads in fact yield better running times than 28 threads on all tested problem instances. Interestingly, the efficiency is more or less constant for both numbers of threads. This could be an indicator that 8 threads will perform better on any input instance, even for very large ones. However, it is not in the scope of this assignment to test on problems where the computational time is more than one and a half hours.

The performance of the parallel algorithm is not as good as expected. This is supposedly due to the repeated access to the shared queues. With 16 queens, the algorithm maintains 8 queues. Thus, with 28 threads, there will always be at least one queue which more than three threads are operating on, i.e., that are pushing new nodes to the queue. Then, using the queues just introduces a lot of overhead.

That the parallel algorithm has a lot of overhead can also be seen from the fact that the speedup is lowest with one thread. The computation of the parallel algorithm with one thread takes more than double the amount of time compared to the serial algorithm.

There is one other fact that prevents the computation from becoming faster with more threads: The way the algorithm is implemented, no thread becomes idle before the queue is empty. However, with 16 queens the first queue contains 16 nodes after initialisation. Thus, the first 16 threads to access the queues will get a node and operate on it. Depending on how long it takes those threads to push new nodes to the next queue, and the other threads to check the queues for emptiness, it may be that the rest of the threads become idle immediately. This should be prevented by adding a variable that holds the number of threads which are currently not working on any nodes. If this counter is equal to the number of nodes, and the queue is empty at the same time, then all the work is done and the threads may become idle.

Quick experiments with this modification showed, that it enhances the performance on the 16-queens problem for more than 8 threads, but only slightly. The remaining factor that the threads all repeatedly access the same critical section weighs in a lot more. One would have to come up with a different algorithm. For example, the number of queues could be even further decreased. This could be done such that configurations are only pushed to a queue after adding more than two queens. Another approach could be to have a fixed number of queues, say three, for every number of queens, and determine how many queens must be added to a configuration at runtime. This would lead to the threads doing more work before accessing the critical sections of the queues, and accessing them less often. However, the bottleneck of accessing the same critical section still remains.

Another approach that uses little to no critical sections could be to use only one queue which is initialised to contain all configurations with two queens. For n queens, there are $(n-1)(n-2)$ valid configurations like this[1]. For large n, this means that there are multiple of those for every thread, even when large numbers of threads are used. Then, the threads could iterate over them and perform the serial algorithm on them. This would reduce the overhead from maintaining the queues, but still split the work evenly enough between the threads.

[1] For the first and the last queen in the first row, $n-2$ spots are valid in the second row. For the other queens $n-3$ spots are valid in the second row: $2*(n-2)+(n-2)(n-3) = (n-2)(n-1)$

5 Conclusion

This work consisted of developing a parallel algorithm for counting the number of solutions for the n-queens problem for a given number of queens. Experiments showed, that the method that was used to split the work between multiple threads introduces too much overhead. For the 16-queens problem, a speedup of 2.16 could be achieved.