**CX 4220/CSE 6220 Introduction to High Performance Computing**
**Spring 2022**
**Programming Assignment 1**
**Due Wednesday February 16**

# 1 Problem statement

The $n$-queens problem is to position $n$ queens on an $n \times n$ chessboard such that no two queens threaten each other. The problem was first published in the German chess magazine Schack in 1948. Let $(i, j)$ and $(k, l)$ denote the respective positions of two queens. The queens are said to threaten each other if

- $i = k$, or
- $j = l$, or
- $|i - k| = |j - l|$

Consider the standard $8 \times 8$ version of the problem. A brute force search algorithm will try 64 positions for the first queen, 63 for the next etc., to result in $1.78 \times 10^{14}$ possibilities, too enormous to enumerate computationally. The search space can be significantly reduced by a few simple heuristics. First, realize that any valid solution must have exactly one queen in each row and one in each column. Thus, a possible solution to the problem can be represented by an array of $n$ numbers. The first number denotes the row position of the queen in the first column etc. The different possibilities can be systematically explored in the following way:

Suppose that the first $i$ entries of the array are already filled and there are no conflicts so far. Then, try each of the $n$ possibilities for the next position. For each possibility, first check if the configuration so far is still valid. If not, reject and move on to the next possibility. If valid, then recursively try all possible ways of filling the remaining columns first, before advancing to the next possibility on the same column. When the array is full and the solution is valid, print the solution. One can come up with more sophisticated strategies to further reduce the search space and you are welcome (but not required) to do so.

# 2 Parallel algorithm

The objective of the programming assignment is to write a parallel program to solve the $n$-queens problem. Your solution should follow the following outline: We use the master-worker paradigm, where processor with rank 0 will act as the master processor and the rest of the processors act as workers. Processor 0 will initiate the search by starting to fill an array of positions as described before. However, whenever a specific depth $k$ is reached, a copy of the array will be dispatched to one of the worker processors. The worker processor will explore all the remaining solutions to

report any solution(s) found to the master processor. As soon as the task is dispatched, the master processor will continue by exploring the next position on the same column etc. to generate another task and dispatch it.

One can view the master processor as performing a search limited to the first $k$ positions. Each worker processor will get a task with first $k$ positions filled and will perform a search for the remaining $n - k$ positions. When a worker is done with its task, it will report the outcome and request for more work. The master processor will continually dispatch work until all the tasks are over. At this stage, it will send a message to all workers to terminate and terminates itself.

Develop two versions of such a parallel program − one that stops as soon as one solution is found, and the other that stops after the entire search space is explored and all solutions are found.

Turn in the following:

1. Your well-commented, easy to read program.
2. A graph plotting the run-time of the program vs. the number of processors for
   - the case where you terminate after finding one solution.
   - the case where you terminate after finding all solutions.

   What observations can you make?

The maximum number of processors in your graph should be at least 16. Use $n = 8$ and $k = 4$. Make sure that your graph presents the data well. It may be beneficial to use a logarithmic scale to better see differences in small values.

## 3 Code framework

In this programming assignment, you will implement the master-worker method to solve the n-queens problem using the framework provided at `https://github.gatech.edu/gbruer3/hpc_sp22`. `Please ensure you clone it as a git repo in case of any updates`. The framework has three core functions that you need to implement:

- `seq_solver`
  Function that implements the serial backtracking solution and produces either all possible solutions, or a single solution, to the n-queens problem.
- `nqueen_master`
  The master repeatedly produces a k length incomplete solution and sends these partial solution available workers as described above. When workers return with a set of solutions, the master stores them and hands this worker with the next available task. If there are no more partial solutions to be sent and all jobs sent to workers have been completed and returned to master, or only a single solution is requested, master will send a kill signal to all workers and quit.

- `nqueen_worker`
  The worker will wait for a message from master. If this message is a partial k length solution to n-queen problem, the worker will finish it and return the complete solution to the master. Note that more than one complete solutions can be obtained from a partial k length solution. If the message received from master is a kill signal, the worker will quit

These functions are declared in `solver.h` and you need to implement them in `solver.cpp`. You can also declare and define any additional helper functions that you might need in `solver.cpp`. Refer to the README.md file for more details and instructions to compile and run the program. Compare your output against the sample outputs provided to check for the accuracy of your solution. *Please* ***do not modify*** *any file in the framework apart from* `solver.cpp/solver.h`.

### 3.1 Output format

A successful run of the program will print out the values $\langle n \ \ p \ \ k \ \ t \rangle$ in a tab-separated format. Here $t$ denotes the time taken by your algorithm. In addition, an output file named *out_n_p_k_e.txt* is produced that contains the solutions for that run. This file contains a number $m$ in the first line, where $m$ is the number of solutions produced by your algorithm, or -1 if only the first solution is requested. This is followed by $m$ (or 1) lines, each line consisting of $n$ numbers representing a solution array, separated by a single space character. A few sample output files are provided with the framework.

### 3.2 Testing

*Testing accuracy*: We have provided some sample outputs in the sample directory (for $n = 6$, $n = 8$, and $n = 10$), which you can use as a reference to compare if your solutions are correct. For grading, we will compare the output of your program for only $n = 8$, but you are welcome to test with the other examples.

The order of solutions can be arbitrary, so for comparison, you can use the bash tools (or equivalent tools if using a different shell) `sort` and then `diff`:

```
diff <(sort your_8.txt) <(sort sample/8.txt)
```

*Testing timing*: When running your code on coc-ice, the timings for a given test case might vary when run multiple times. To account for that, we encourage you to run a test case multiple times and use a collective measure to report your final time (for e.g. median or average over all runs, eliminating an outlier case). However, do not force your timings to follow an "expected" curve. Use a scientifically reasonable measure to evaluate multiple timings for a test case and mention the measure you use in your report.