

PPA-F26 hw1

Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below.

Submission instructions will be announced soon. (We are working on setting up a Gradescope instance for the course; it should be ready soon.)

Create a file `written.pdf` containing your speedup plots.

For the coding tasks, you will edit and submit the file `NQueensExt.sml`.

DUE DATE: This submission is due at **5:00pm EDT on Monday, Sep 21**. Course policy on late submissions is available on the course website. (<https://cs.nyu.edu/~shw8119/courses/f26/3033-121-ppa/>)

Setup

- Follow the instructions on the course website to access one of the `crunchy` compute servers. See more info [here](#).
- Install MaPLE. Locally on your own machine, we recommend using `mpl-switch` and installing version `ca92f64`. For the `crunchy` servers, you can do the following:
 - Download <https://cs.nyu.edu/~shw8119/courses/f26/3033-121-ppa/resources/mpl-ca92f64.tgz> and copy it to the server
 - Unpack by running `tar xzf mpl-ca92f64.tgz`. This will create a directory `mpl-ca92f64/`
 - The compiler, `mpl`, is located at `mpl-ca92f64/bin/mpl`
 - Add `mpl-ca92f64/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-ca92f64/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

A Fun Place to Start: N-Queens

The [N-Queens problem](#) is a fun mathematical curiosity which asks: how many different ways can N queens be placed on an $N \times N$ chessboard, such that no queen can take another in a single move?

This is a nice “hello world” problem for parallel computing, so let’s play with it.

To count the number of solutions, we could use a brute-force approach. The idea would be to consider all possible placements of N queens and check to see in each case if any queen is threatened. This approach can be naturally parallelized because each potential placement of queens can be checked independently.

To be a bit more efficient, rather than generating *all* possible placements of queens on the board, we can instead attempt to consider only the placements that have *at most one pair* of threatened queens.

The idea is to start with an empty board and then generate N children, each of which has a queen in a different column of the *first* row. Each of those children then has N more children, where we attempt to place a queen in each column of the *second* row, and so on. Whenever a potential placement has a threatened queen, we don’t need to consider any of its children.

For example, here is picture showing the beginning of this algorithm for $N = 3$. The example shows 7 boards, and 2 of these have threatened queens, so we don’t need to consider any of their children. For each of the “...” in the picture, the search is incomplete.

A Parallelized Search

Take a look at the file `NQueens.sml`. The function `count_solutions(size)` in this file implements the algorithm described above and counts the number of solutions (where `size = N`). It does so by creating tasks to search through the children of every board in parallel, and adding up the results.

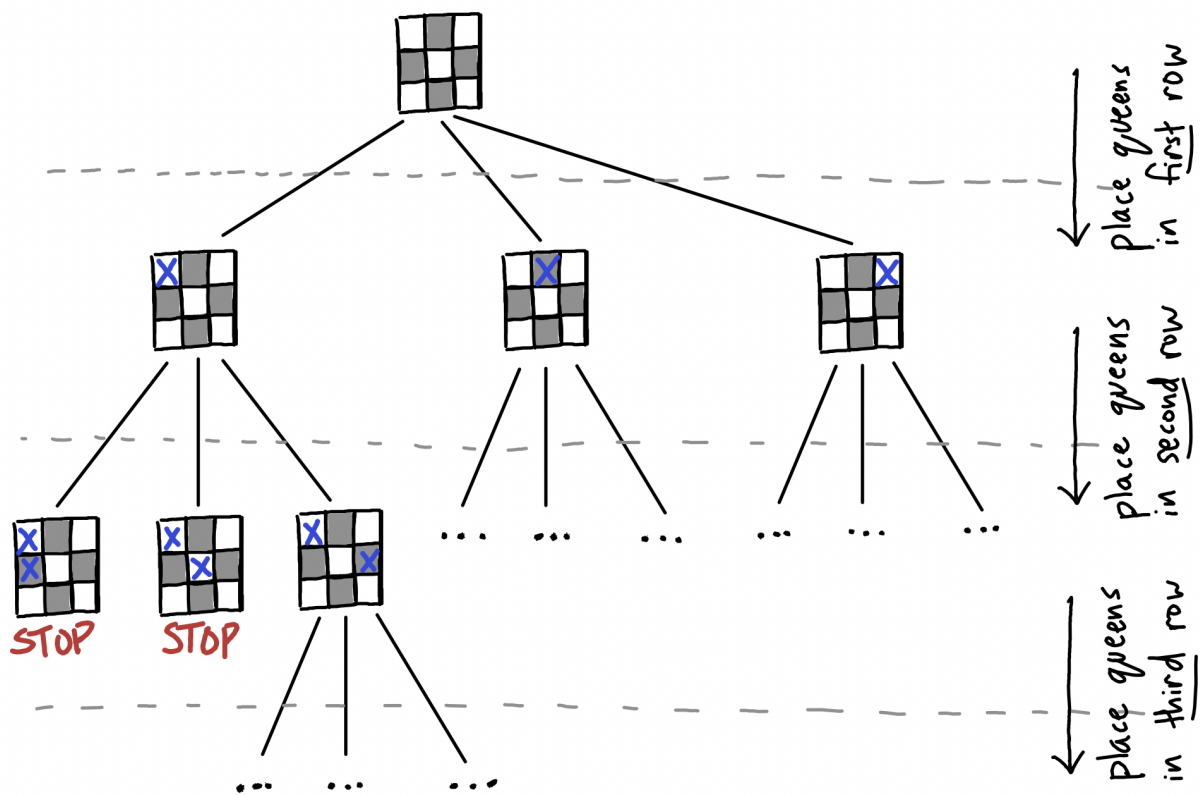


Figure 1: Search tree for 3-Queens

Task 1 (10 points): Compile, run, and create a self-speedup plot for this code using 1-32 processors on input size $N = 13$. See the instructions below. Each run will report the average run-time across 5 repetitions and will print out the number of solutions. Use the command-line arguments `@mpl procs <P> --`, replacing `<P>` with the desired number of processors. For $N = 13$, it should say there are 73,712 solutions.

```
$ make main
$ ./main @mpl procs 1 -- -n 13 -warmup 1 -repeat 5
$ ./main @mpl procs 2 -- -n 13 -warmup 1 -repeat 5
# ... and so on ...
```

You can run this command to automatically step through every processor count between 1-32 and only report the average:

```
$ for p in $(seq 1 32); do \
    echo -n "procs $p "; \
    ./main @mpl procs $p -- -n 13 -warmup 1 -repeat 5 | grep average; \
done
```

Let $T(P)$ be the amount of time it takes to run on P processors, and define $selfspeedup(P) = T(1)/T(P)$. **Your task is to make a self-speedup plot, where the X-axis is the number of processors (ranging from 1 to 32), and the Y-axis is $selfspeedup(P)$.**

(Note: Do this on a machine that has at least 32 processors! We recommend the [crunchy servers](#); see more info [here](#))

(Note: feel free to make the plot however you'd like. Not a bad use-case for AI to write the plotting script...)

A Modified Search

The code in `NQueens.sml` only counts the number of solutions. Your next task is to create a new search which not only counts the number of solutions but also counts the number of boards considered.

Task 2 (30 points): In `NQueensExt.sml`, implement the function `count_solutions_and_boards(size)`. It should return a tuple (n_s, n_b) , where n_s is the number of solutions and n_b is the number of boards considered by the search.

Don't modify `NQueens.sml`. We recommend copy/pasting code from `NQueens.sml` into `NQueensExt.sml` and then modifying it as you see fit. As a hint, consider implementing a new helper function `parallel_sum_tuples(lo, hi, f)` which expects the function `f(...)` to return a tuple of integers and sums them up.

To do a bit of unit testing, you can run the following. Feel free to add more tests at the top of `test-ext.sml`.

```
$ make test-ext
$ ./test-ext
```

You can test the performance of your code as follows, using the `--ext` flag to tell the benchmark drive to run your `NQueensExt.sml` code instead of `NQueens.sml`.

```
$ make main
$ ./main @mpl procs 8 -- -n 13 -warmup 1 -repeat 5 --ext
```

Task 3 (10 points): Similar to Task 1, create a self-speedup plot for your new `count_solutions_and_boards(size)` function. You can run this command to automatically step through every processor count between 1-32 and only report the average:

```
$ for p in $(seq 1 32); do \
    echo -n "procs $p "; \
    ./main @mpl procs $p -- -n 13 -warmup 1 -repeat 5 --ext | grep average; \
done
```

Mirages

Notice that the search sometimes does a lot of work only to come up empty-handed. In the search tree for $N = 3$, for example, none of the boards lead to a solution, and yet the search still considers every board in the tree.

Let's give this phenomenon a name. Consider a partial board that has one queen in each of the first k rows (for some $0 \leq k \leq N$), none of which are threatened. This board is a node in the search tree, and the *search tree beneath it* consists of the board itself together with all of its descendants in the search tree. The *size* of this subtree is the number of boards in it, i.e., the number of boards the search would consider if it started from this board. (This is exactly the quantity n_b from Task 2, but for a search that starts from the partial board instead of from the empty board.)

We say that such a partial board is a **mirage** if there are no solutions anywhere in the search tree beneath it. A **maximal mirage** is a mirage whose search tree is as large as possible: among all mirages for a given N , it has the largest number of boards beneath it. In other words, a maximal mirage is the biggest waste of effort in the whole search.

For example, for $N = 3$, the empty board is a mirage, and it is the maximal mirage: the number of boards beneath it is the size of the entire search tree. For $N = 1$, there are no mirages at all.

Task 4 (40 points): In `NQueensExt.sml`, implement the function `maximal_mirage(size)`. If there are no mirages, it should return `NONE`. Otherwise, it should return `SOME (board, n_b)`, where `board` is a maximal mirage (of type `ChessBoard.board`) and n_b is the number of boards in the search tree beneath it (including the mirage itself).

For example, at $N = 3$, the empty board is a maximal mirage, with $n_b = 19$.

If there are multiple maximal mirages, you may return any one of them.

Your implementation should be parallel, and it should find the maximal mirage in a single pass over the search tree (i.e., do not run a separate search from every partial board!). As a hint: think about what information you need to return for each subtree so that the results from the children of a board can be combined into a result for the board itself. Notice that if a subtree contains no solutions, then its root is a mirage, and it is at least as large as any other mirage inside that subtree; otherwise, the largest mirage in the subtree is the largest among the mirages found by the children.

Similar to Task 2, you may find it helpful to copy/paste the `parallel_sum` function and modify it or generalize it to work over different types of data.

The file `test-ext.sml` includes a few tests for `maximal_mirage`. For each board size, the tests list every maximal mirage, and check that the board you return is one of them and has the expected number of boards beneath it.

```
$ make test-ext
$ ./test-ext
```

You can also run your code on a larger board and print out the mirage it finds, using the `--mirage` flag:

```
$ make main
$ ./main @m1 procs 8 -- -n 13 -warmup 1 -repeat 5 --mirage
```

Task 5 (10 points): Similar to Tasks 1 and 3, create a self-speedup plot for your `maximal_mirage(size)` function on input size $N = 13$. You can run this command to automatically step through every processor count between 1-32 and only report the average:

```
$ for p in $(seq 1 32); do \
    echo -n "procs $p "; \
    ./main @m1 procs $p -- -n 13 -warmup 1 -repeat 5 --mirage | grep average; \
done
```