Objective
The objective of this assignment is to implement a parallel shared-memory version of the well-known \texttt{N-Queens} problem from scratch. You will again use the ANL M4 macros. You will have the opportunity to run and evaluate your shared-memory program on an 8-CPU machine.

Problem description: The N-Queens Problem
The well-known \texttt{N-Queens} problem asks you to find out all possible ways of putting $N$ queens on an $N \times N$ chessboard such that no two of them attack each other. A queen can attack another if it is on the same row, the same column or the same diagonal.

In this assignment the problem that you are going to solve is a slightly modified version of the \texttt{N-Queens} problem. We assign a profit to each of the squares of the chessboard. The $(i,j)$ entry has a profit of $|i-j|$ for $0 \leq i \leq N-1$ and $0 \leq j \leq N-1$ (use \texttt{abs} function of C to calculate the absolute value). Thus if you place a queen on the $(i,j)$ square you gain $|i-j|$ units. In this assignment, you will parallelize this problem and write a shared-memory version of it. Your program should take two parameters from the command line, namely, $N$ and $P$. The program on termination should print out the total number of solutions found, the solution that maximizes the profit (find a good way to print this solution) and the profit corresponding to this best solution. Also it should print out the timing breakdown i.e. initialization time, computation time, and finish time. Use the same methods for measuring timing statistics as done in PA1.

Setup
This programming assignment should be conducted as in PA1. Use the ANL macros \texttt{(c.m4.ece5750)}, lock implementation \texttt{(lock.s)}, and \texttt{Makefile} from the previous assignment to write and compile your program, and \texttt{qsub} to submit it to the queue for execution on the 8-CPU machine.

Production runs on the 8-CPU machine
Once you have tested your program to run correctly on your local machine, you should execute the following steps to submit your program for execution on the 8-CPU machine. These are the same steps as in PA1, and are given here for your convenience.

1. Create a file named \texttt{run} that contains the following two lines (replace \texttt{foo} with...
your executable name, and \(N\) and \(P\) with your program’s arguments).

```bash
#!/bin/sh
./foo N P
```

Do not try to include multiple commands in the `run` file, as only the first command will be considered and the rest would be ignored.

2. `chmod 755 run`  
3. By default, your program result will be sent to your Cornell e-mail. If you would like to use another account, create a file named `email` that contains a single line of a complete e-mail address.
4. `zip pa.zip foo run [email]`  
   `email` is optional, as described in (3)
5. `qsub pa.zip`

The queuer daemon will send you an e-mail containing the output of your program. If you receive an empty e-mail try to find out if you did anything wrong. Before executing the fifth step I highly recommend that you create a new directory, unzip `pa.zip` in that directory and execute `./run`. If that works satisfactorily then only proceed to execute the fifth step. This is very important because if you do something that is very wrong it may unnecessarily block the queuer.

**Submission**

You must submit a **correct** parallel implementation of the algorithm described. You are allowed to use the web or any other consulting resource to find the best possible approach; however, you **must properly cite all your references**. Your solution should be shared-memory and use the ANL M4 macros where appropriate.

You must also submit a two- to four-page report (PDF preferred; Word or plain text acceptable) of your implementation, in which you explain:

- **The Algorithm:** what algorithm you used to parallelize the problem and the rationale behind it. Discuss how you partitioned the task and the trade-offs involved.

- **Experimental results:** prepare a table showing the number of solutions and the maximum profit achievable for \(N\) increasing from 8 to 15. Also list the solutions that achieve the maximum profit for these cases. Next, report (in a meaningful format) the total time and computation time respectively for \(N=8, 9, \ldots 15\) and \(P=1, 2, 4, 8\) and the computation speedup for \(N=8, 9, \ldots 15\) and \(P=2, 4, 8\).
Discussion: Comment on the results regarding the speedup and discuss whether they agree with the theoretically expected results. Also, include a discussion on the shortcomings and possible improvements of your implementation.

Please notice that the submission period ends automatically at 11:59pm on the day of the submission.

Start early!! Good luck!

Extra Credit
The following should only be attempted after finishing your N-Queens code above. Your efforts can add up to 25 points. The assignment is to write a parallel shared-memory version of Latin Squares. In the Latin Squares problem there is an $N \times N$ matrix and $N$ colors. The goal of the program is to color the matrix so that a color appears only once in each row and each column. Clearly, if you started with an empty (un-colored) matrix, the problem is trivial. Similarly, given a pre-colored matrix with $S$ set (colored) cells and some "holes", when starting with only a small number of holes ($S$ is close to $N^2$) it is easy (quick) to determine whether the matrix is colorable or not. The difficult cases (and long running!) are when $S = \frac{N^2}{2}$ but can vary substantially.

Your first task is to come up with a scheme to generate random solvable instances of the problem with a parameterizable $S$ (you may want these random instances to be repeatable for testing purposes). Then for $S=0.3x, 0.4x,$ and $0.5xN^2$ solve as large a problem ($N$) as possible in under 2 minutes on an eight-processor machine. Note that you can test this first by running your uniprocessor version and getting it close to 8 minutes (if you want to be optimistic about your parallel version). In your report, state how large a problem you were able to run, how long it took, and outline the algorithm and any design decisions you care to share. There are many possible choices of matrix representation, color data structures, etc. It has been observed that randomness helps the convergence of this problem, so you may want to keep that in mind when making some color choices.