CX 4220/CSE 6220 Introduction to High Performance Computing
Spring 2019
Programming Assignment 2

Submission guidelines

We will be using automated scripts for grading so follow the instructions carefully.

1. The assignment is due on **March 7, 11:59 PM EST** on Canvas. You can form a team of upto 3 members.
2. Download the code framework:

   git clone `git@github.gatech.edu:rnihalani3/2019_HPC_prog2_framework.git`

   Refer to section 3 on how to use the framework.
3. Run your solution on the PACE cluster using the PBS script provided in the framework and perform necessary experiments. Present analysis in a report.
4. Add a text file `team.txt`, containing the names of all team members to root directory `prog2_framework` and zip it (`zip -r <zipname>.zip prog2_framework/`).
5. Submit two files: The zipped source file (mentioned in previous point) and a report. Name the zip file and the report with the GT user name of the members of your team `GTusername1_GTusername2_GTusername3.{zip,pdf}`. Don't put your report in zip file.
6. Only one member from each team should upload the zip file and the report on Canvas.
7. **Not abiding by the submission instructions will cause you to lose up to all of the points.**

1 Problem Statement

This assignment requires you to develop a parallel back-tracking algorithm to the classic n-queens problem. The problem was first published in the German chess magazine Schack in 1948. Consider a $n \times n$ chessboard on which n queens are to be placed. 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

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

The n-queens problem is to position the n queens on the chessboard such that no two queens threaten each other.

Note that any valid solution will have exactly one queen in every row and exactly one queen in every column. The complexity of finding a solution is approximately $O(n!)$, which even for small values of $n \sim 20$ can be formidably large. A valid solution to the problem can be represented by an array of n numbers, $sol[0..n-1]$, where $sol[i]$ gives the row in which the queen is present in column i. The rows and columns on the chessboard are numbered from 0 to $n-1$. The back tracking algorithm is used to find a valid solution by recursively finding valid configurations $sol[0..i]$ from $sol[0..i-1]$. This can be thought of as searching depth-first in a tree of configurations until we reach a valid leaf node, $sol[0..n-1]$. Particularly, back tracking can be implemented as follows -

Consider a sub problem S_i where $sol[0..i-1]$ is filled without any conflicts and a queen in the i^{th} column needs to be placed. We try each of the n possibilities for $sol[i]$. For each possibility, check if the queen placement so far is still valid (i.e. queen in column i is not threatened by queens in columns $\{1 \ldots i-1\}$). If invalid, reject and move on to the next possibility for $sol[i]$. If valid, then consider the sub problem S_{i+1} of placing the queen in $i+1^{th}$ column given a valid $sol[0..i]$ before moving on to the next possible position in column i. That is, try all possible ways of filling columns $\{i+1 \ldots n-1\}$ before advancing to the next possibility on column i. When the array is full and the solution is valid, report 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

A parallel implementation of the backtracking algorithm descried can be done using the **master-worker** paradigm. On a p processor architecture, one processor (rank 0) acts as the master and the remaining processors act as workers. The master will initiate the backtracking search by filling the solution array as described before. However, whenever a specific number of positions k are filled, a copy of this partially filled array will be dispatched to one of the worker processors. The worker processor will perform the task of exploring all the remaining solutions to this sub problem S_k and report any solution(s) found to the master processor. As soon as the master dispatches a task to a worker, the master processor will continue by exploring the next configuration with the first k positions filled to generate another task and dispatch it to a worker.

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 wait for more work. The master processor will continually dispatch tasks to *available* workers. A worker is available if it is currently not performing any task (all workers are available in the beginning). When all workers are unavailable, the master waits for a worker to return with solution(s) and sends a new task to that newly available worker. This is done until all the tasks are over. At this stage, the master will send a message to all workers to terminate and terminates itself.

3 Framework

In this programming assignment, you will implement the master-worker method to solve the n-queens problem using the framework provided. The framework has three core functions that you need to implement:

- `seq_solver`
  Function that implements the serial backtracking solution and produces all possible solutions 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, 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`*.

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.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. This is followed by m 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 $n = 6, n = 8, n = 10$), which you can use as a reference to compare if your solutions are correct. For grading, we will compare the output of your program with these and other, much larger instances.

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. 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.

4 Grading criteria

Your program will be graded on the following criteria:

- *Efficient implementation*: This would be an important criteria in grading your submission. You should implement the three functions in an efficient manner and test the scalability of your parallel algorithm.
- *Computation-communication overlap*: Consider the situation where all workers are busy (the master is waiting) and a worker W has just returned with its task finished, ready to be handed the next task. There are two ways to implement this
  - After the master receives solutions from W, it then creates the next partial solution (while letting W stay idle), and then hands it over to W.
  - Alternatively, the master can overlap the partial solution computation with the waiting time. More specifically, instead of just waiting and doing nothing, master can create the next partial solution and have it ready. The moment W returns, the master immediately hands the already prepared partial solution to W, eliminating W's idle time. This can be achieved by using the non-blocking MPI calls instead of blocking.

  While both methods are accurate, the latter is more efficient and a few points will be reserved for implementing this correctly.
- *Accuracy of the output.*
- *Code readability*: Your implementation should be well commented and easy to read.
- *Report*: Give a high level description of your algorithm for the three functions (Do not go into low level implementation details). Report any ideas or optimizations of your own that you might have applied and how that might have affected the performance. Test your implementation on various various values of n, p and k on the `coc-ice` cluster. We will have our own test cases. Vary one parameter while keeping the other two fixed, and plot the effects of the varying parameter on runtime (thereby giving at least 3 plots, one for each varying parameter). When varying p, choose an appropriate n (not too low that you don't observe any interesting scalability, and not too high that your serial algorithm doesn't finish in time alloted by the cluster). When varying k, choose an $n > 10$ and vary k from $\{1 \dots n\}$. What k value gives you the minimum runtime? Explain why. When varying n, choose $p \geq 8$. What largest n can you solve with $p = 16$? What k enables you to solve for that largest n?

You may address your own interesting questions in addition to the above mentioned, and you will get extra credit for that. Ideally, your report should not exceed 3 pages (though no points will be deducted if you need to add more pages).