Lab 3: Finish Accumulators and NQueens Problem

Instructor: Vivek Sarkar, Co-Instructor: Mackale Joyner

Course Wiki: http://comp322.rice.edu

Staff Email: comp322-staff@mailman.rice.edu

HJlib Documentation: http://pasiphae.cs.rice.edu/

Goals for this lab

- Implement a parallel version of NQueens using Finish Accumulators, and evaluate its performance using abstract metrics (`Lab3NQueensAbstractMetricsCorrectnessTest`)
- Extend the implementation for improved real performance (`Lab3NQueensHjPerformanceTest`) by using a "cutoff" strategy.

Downloads

As with previous labs, the provided template project is accessible through your private SVN repo at:

`https://svn.rice.edu/r/comp322/turnin/S17/NETID/lab_3`

For instructions on checking out this repo through IntelliJ or through the command-line, please see the Lab 1 handout. The below instructions will assume that you have already checked out the lab_3 folder, and that you have imported it as a Maven Project if you are using IntelliJ.

1 The N-Queens Problem

Today's lecture on Finish Accumulators introduced the N-Queens problem, viz., how can we place N queens on an N × N chessboard so that no two queens can capture each other? This problem was also presented in the demonstration video for topic 2.3.

You will edit the `NQueensHjLib.java` file provided in your svn repository for this exercise. There are `TODO`s in this file guiding you on where to place your edits. In IntelliJ, you can automatically find all TODOs by going to View > Tool Windows > TODO.

The lab code already contains a sequential implementation for solving the N-Queens problem (`NQueensSequential`). The first goal of this lab is to create a parallel version of an N-Queens solver using HJlib and finish-accumulators, as described in today's lecture. These changes should be made in the `NQueensHjLib` class, primarily in the `nqueensKernel` method. Your solution to this part should be tested by only running `Lab3NQueensAbstractMetricsCorrectnessTest`; we will focus on `Lab3NQueensHjPerformanceTest` in the next part of the lab.

For convenience, the implementation of the `okToPlace()` method in the `NQueensRunner` class performs one call to `doWork(1)` for each pair of squares that is tested. This is the work that feeds into the WORK and CPL values generated by the abstract metrics, and so you should re-use the `okToPlace` method in your parallel implementation based on how it is used in `NQueensSequential`.

```
void foo (int depth) { // Assume WORK decreases as depth increases (
    if (n == LIMIT ) {
        // Process leaf case
        . . .
        return;
    }

    if (n < CUTOFF) { // PARALLEL VERSION
        . . .
        async foo(n+1);
        . . .
     }
     else { // SEQUENTIAL VERSION
        . . .
        foo(n+1);
        . . .
     }
}
```

Figure 1: Code schema for parallel divide-and-conquer algorithm with cutoff

2 A Note on Real Performance and the Cutoff Strategy

2.1 Real Performance

In this lab, we move from abstract metrics to real multi-threaded performance for the first time. Real performance is messier than abstract performance because it is affected by its environment. Many students will have varying hardware, varying software, and possibly other applications running at the same time when doing this lab on their laptops. Even your laptop's power manager can throttle the number of hardware cores your HJlib program gets to use, limiting your speedup when not plugged into a wall socket.

In later homeworks and labs, we will teach you how to run parallel programs on Rice's compute clusters. (This is why you may have been notified about receiving an account on Rice's NOTS system.) Compute clusters are professionally managed and deployed machines designed to ensure that running applications do not see interference from others running in the same cluster.

However, for today's lab, you should start by simply seeing what real-world speedup you can achieve on your local laptop. If you do not see any speedup in real performance (i.e., if you cannot pass `Lab3NQueensHjPerformanceTest`), you can try closing down any other expensive applications that might be running, or plugging your laptop into a power supply. Feel free to call over a TA for help, they can help judge if the problem is in the code or the environment. You will not be penalized if your code fails `Lab3NQueensHjPerformanceTest` on your laptop due to environmental factors.

You should also try submitting your solution to the autograder. The autograder will automatically handle transferring your solution to one of Rice's clusters and executing it there. This guarantees that you will see speedup with a correct solution, and consistent results from one run to the next. You can also use the autograder's Leaderboard feature to see what performance other students are getting on the same tests and platform.

2.2 Cutoff Strategy

A common way to reduce the overheads seen in real performance when creating large numbers of tasks (each of which does very little work) is for the programmer to add a *cutoff test*. Figure 1 shows a code schema for a parallel divide-and-conquer algorithm with a CUTOFF value. While conceptually `async`s do not do

any actual application work, the creation of an `async` still consumes both CPU and memory resources. As a result, while creating an excessive amount of `async`s might maximize the abstract parallelism of your application, it may actually lead to your code running slower than a sequential implementation.

In the first part of this lab, you developed a parallel implementation of NQueens that passed the abstract performance tests in `Lab3NQueensAbstractMetricsCorrectnessTest`. In this section, we will use the cutoff strategy to implement a more efficient parallel implementation that can also pass the real world performance tests in `Lab3NQueensHjPerformanceTest`.

As a first step, it might be interesting to run the tests in `Lab3NQueensHjPerformanceTest` on your existing parallel solution that does not use the cutoff strategy. You will probably find that the performance of your parallel implementation actually runs much more slowly than the provided sequential implementation. For example, sample experiments on a laptop with four cores yield speedups between 0.1× and 0.2× the sequential implementation, meaning that the parallel implementation is running 5-10× slower! (Note that the tests in `Lab3NQueensHjPerformanceTest` do not fail if your code runs slowly, but simply print out the produced speedup).

Instead, we can implement the cutoff strategy by saying that for any `depth` greater than or equal to a certain cutoff, we will run the remaining computation sequentially. Note that `depth` is a parameter passed to `nqueensKernel` that increases by one on each recursive call. You can access a recommended cutoff value for the current test by calling `getCutoff()` inside of `NQueensHjLib`, or you can use a suitable constant that you choose as the cutoff.

Once you have implemented the cutoff strategy, try re-running the tests in `Lab3NQueensHjPerformanceTest` on your laptop to see if the results have improved from your initial, maximally parallel solution. You should also try submitting your solution to the autograder, which will run these performance tests on one of Rice's compute clusters. While your laptops may have four or eight cores, this particular cluster has 16 cores and so you may expect to see much larger speedups running there.

3 Demonstrating and submitting in your lab work

For this lab, you will need to demonstrate and submit your work before leaving, as follows.

1. Show your work to an instructor or TA to get credit for this lab. They will want to see your files submitted to Subversion in your web browser and the passing unit tests on your laptop or on the autograder.
2. Check that all the work for today's lab is in your `lab_3` directory by opening https://svn.rice.edu/r/comp322/turnin/S17/NETID/lab_3/ in your web browser and checking that your changes have appeared.