Lab 6: Barriers and Phasers

Instructor: Vivek Sarkar

1 Update your HJ/DrHJ Installation

The performance measurements for today's lab should be done on Sugar, and we've already updated the HJ installation there. (See Lab 4 handout on instructions to access the HJ installation in the COMP 322 userid on Sugar.)

However, if you're also working with a local installation, please update it from the HJ download page, `https://wiki.rice.edu/confluence/display/PARPROG/HJDownload`, to make sure that you have the latest updates and bug fixes.

2 One-Dimensional Iterative Averaging Example: Forall version

The code in `OneDimAveraging.hj` performs the iterative averaging computation discussed in the lectures. This code performs a sequential version of the computation in method `runSeq()` and a parallel chunked `for-forall-for` version in method `runChunkedForkJoin`, as discussed in Lecture 10.

The input arguments for the main method in this program are as follows:

1. `tasks` = number of chunks to be used for chunked parallelism. The default value for `tasks` is `Runtime.getNumOfWorkers()`, which is the number of workers w specified with the "`-places` $1:w$" option (default is $w = 8$ on SUGAR).
2. `n` = problem size. Iterative averaging is performed on a one-dimensional array of size `(n+2)` with elements 0 and `n+1` initialized to 0 and 1 respectively. The final value expected for each element i is $i/(n+1)$. The default value for n is 100,000.
3. `iterations` = number of iterations needed for convergence. The default value is 20,000. This default was set for expediency. For this synthetic problem, you typically need $O(n^2)$ iterations to guarantee convergence, which would be 10^{10} for $n = 100,000$. The default of 20,000 was selected so that the runs can be completed quickly in a single lab session.
4. `rounds` = number of repetitions for the entire computation. As discussed earlier, these repetitions are needed for timing accuracy. The default value is 5. For 5 repetitions, a reasonable approach is to just report the minimum time observed.

You should run your program on SUGAR, to evaluate the parallelization. As before, you can compile the program as follows, after repeating the setup from Lab 4:

```
hjc OneDimAveraging.hj
```

To run the program using 8 cores, use the following command on a *compute node* (obtained using the "qsub -I ..." command discussed in Lab 4):

```
hj -places 1:8 OneDimAveraging
```

Your task for this section is to record the best sequential and chunked-forall times observed for the default inputs (using 8 cores), and then compute their ration as the speedup.

Please pay special attention to understanding the `getChunk()` method used in this code. The method signature is slightly different from the version discussed in the lecture, but the functionality is the same. You

will need this method for the tasks in the next section. You are also welcome to use this method in your homework solutions.

3 One-Dimensional Iterative Averaging Example: Phaser versions

In this section, your task is to write and evaluate the following phaser variants of the parallel forall computation studied in the previous section:

1. **Barrier version:** The code in method `runChunkedForkJoin` corresponds to the code studied in slide 8 of Lecture 13. Create a modified version of this method that uses barrier synchronization for efficiency, as discussed in slide 9 of Lecture 13. Compare its performance with that of the fork-join version on SUGAR.
2. **Single statement:** Extend the previous version with a next-single statement that performs a print operation as discussed in slide 11 of Lecture 13. You need not evaluate the performance of this version.
3. **Phaser version with point-to-point synchronization:** In slide 16 of Lecture 14, we discussed how the use of point-to-point synchronization may be more efficient that phasers for the Iterative Averaging example. Create a modified version of the `runChunkedForkJoin` method that uses point-to-point phasers instead of a barrier, and compare its performance with that of previous versions on SUGAR.

4 Tips and Pitfalls

Here are some programming tips and pitfalls to be aware of when working on today's lab:

- **`phaserMode` prefix:** Be sure to include the `phaserMode` prefix when referring to phaser registration modes in code e.g., `phaserMode.SIG`, `phaserMode.WAIT`, etc. This is because `phaserMode` is a predefined `enum` in the `hj.lang` package.
- **Local pointers to shared objects:** Pay careful attention to pointers `myVal` and `newVal` in slide 9 of Lecture 13. These pointer variables are local to each `forall` iteration that point to the same shared objects (arrays). Note that the pointer swap is performed redundantly for each pair of local variables. There is little overhead in performing this computation redundantly, because only pointers are swapped. In contrast, performing the swap only once would require synchronization which could result in incur overhead.
- **Performance benefits of point-to-point vs barrier synchronization:** There are multiple factors that determine whether or not point-to-point synchronization will result in better performance than the barrier version. First, it depends on the variability in the computations. If the amount of work performed in each phase is nearly equal, there isn't a big difference in the CPL of the two versions, even though the point-to-point version imposes fewer constraints than the barrier version. On the other hand, if the amount of work performed by `forall` iterations varied from phase to phase, the point-to-point version is likely to perform better. Second, the cost of a barrier increases with the number of workers/processors performing the barrier. You may not see much of a difference between the two versions on a Sugar compute node with 8 workers/processors, but the difference will be larger with (say) 32 or 128 workers.