Lab 5: Real Performance, Work-Sharing and Work-Stealing Schedulers

Instructor: Vivek Sarkar

Resource Summary

Course wiki: `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`

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

Coursera Login: visit `http://rice.coursera.org` and log in via Shibboleth

Clear Login: ssh *your-netid*@ssh.clear.rice.edu and then login with your password

Sugar Login: ssh *your-netid*@sugar.rice.edu and then login with your password

Linux Tutorial visit `http://www.rcsg.rice.edu/tutorials/`

IMPORTANT: please refer to the tutorial on Linux and SUGAR, before staring this lab.

1 One-time Setup on SUGAR

You should have an account on SUGAR (`http://www.rcsg.rice.edu/sugar`), which is a cluster of Intel Xeon machines, similar to CLEAR. The main difference is that SUGAR allows you to gain dedicated access to compute nodes (see qsub command below) to obtain reliable performance timings for your programming assignments. On CLEAR, you have no control over who else may be using a compute node at the same time as you.

- Login to SUGAR.
  ssh ⟨your-netid⟩@sugar.rcsg.rice.edu
  ⟨your-password⟩
  You should have received an email with the default password convention for Sugar. Use that password when you login the first time. You will be asked to set a new password immediately. Note that this login connects you to a *login* node.
- On SUGAR, HJ is already installed at /users/COMP322/hj/bin. Run the following command to setup the HJ path.
  source /users/COMP322/hjsetup.txt
- Check your installation by running the following commands:
  which hjc
  You should see the following: /users/COMP322/hj/bin/hjc
  which hj
  You should see the following: /users/COMP322/hj/bin/hj
- When you log on to Sugar, you will be connected to a *login node* along with many other users. To request a dedicated *compute node*, you should use the following command from a SUGAR login node:
  qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
  When successful, it will give you a command shell on a dedicated 8-core compute node for your use for 30 minutes at a time. Your home directory is the same on both the login and compute nodes.

  For now, just type "*exit*" immediately after you obtain the compute node. We will repeat this command later.

We only have 14 nodes available for dedicated use for COMP 322. When this limit is exceeded, your request for compute nodes will be pooled with other requests at Rice, which may result in delays. Therefore, it is *very* important that you restrict your use of compute nodes to performance timings. General edit, compile, and debug of HJ programs can be done on any other computer, or on the SUGAR login nodes. However, please make sure that you don't run any long jobs (> 1 minute) on a SUGAR login node.

2 Timing your program execution

1. (FYI) To measure the execution time of your program, you should insert timing calls to get the system time in nanoseconds before and after the computation that you want to measure. The difference between the two gives the actual time spent on the computation in nanoseconds.
   long start = System.nanoTime();
   // Computation that you wish to time
   long end = System.nanoTime();
   long timeSpent = end - start;
   The time spent in executing a program can vary depending on a lot of factors that are system dependent. Follow the steps below to reduce the variations and time your programs accurately.
   - Repeat the computation in your program multiple times (at least 5 times).
   - Calculate the time needed for the computation in each repetition.
   - Report the minimum of these observed times.
2. Download the `nqueens.hj` program by typing "*cp -p /users/COMP322/examples/nqueens.hj .*" (the period after nqueens.hj is significant).
   (The wget command, *wget* `http://www.cs.rice.edu/~vs3/downloads/nqueens.hj`, will also get the same file.)
3. Compile the program for the work-sharing scheduler (default).
   hjc nqueens.hj
4. Obtain a dedicated compute node before doing timing measurements
   qsub -q commons -I -V -l nodes=1:ppn=8,walltime=00:30:00
5. Run the program on 1 worker, and note the execution times obtained.
   hj -places 1:1 nqueens
   (NOTE: the work-sharing scheduling may increase the number of workers each time a blocking operation is performed. This is not a major issue for nqueens, since it contains only one finish construct.)
6. Run the program on 8 workers (default on SUGAR), and note the execution times obtained.
   hj -places 1:8 nqueens
7. Release your dedicated compute node by typing *exit*. You're now back on the login node.
8. Examine the timing calls in the `nqueens.hj` program, and correlate them with the output that you see. Also, compare the times obtained for 1 vs. 8 workers, and estimate the speedup that you obtained.

3 Experimenting with the async seq clause on the work-sharing scheduler

The *async seq* clause is used in nqueens.hj to limit parallelism when *depth >= cutoff_value*. The default cutoff is 3, but it can be changed via command line arguments. For example, to run the program with 8

workers and a cutoff value to 4, type:
hj -places 1:8 nqueens 12 5 4
since the default arguments were "12 5 3".

Rerun the program with 8 workers with different cutoff values in the range $0 \ldots 12$ and see which one yields the best time for 8 workers and a work-sharing scheduler. You can also experiment with the ForkJoin variant of the work-stealing scheduler, by issuing the following command:
hj -fj -places 1:8 nqueens 12 5 4

As in past labs, create a text file named `lab_5_written.txt` *in the* `lab_5` *directory, and enter your timings and observations there.*

4 Experimenting with the work-stealing scheduler (help-first and work-first policies)

The *nqueens.hj* program belongs to the HJ subset (basic async, finish, atomic operations, isolated) supported by work-stealing schedulers. Recall that there are two main variants of the work-stealing schedulers: help-first (*-rt h*) and work-first (*-rt w*).

To compile and execute the nqueens program with the help-first variant of the work-stealing scheduler, type the following commands:
hjc -rt h nqueens.hj
hj -places 1:8 nqueens 12 5 4

Likewise, to compile and execute the nqueens program with the work-first variant of the work-stealing scheduler, type the following commands:
hjc -rt w nqueens.hj
hj -places 1:8 nqueens 12 5 4

Now, experiment with different cutoff values and the help-first vs. work-first work-stealing variants to try and obtain the best possible time for nqueens. Enter your results in `lab_5_written.txt` .

5 One-Dimensional Iterative Averaging Example

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-finish-forasync–for` version in method `runChunkedForkJoin`, as discussed in Lecture 11.

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:

```
hjc OneDimAveraging.hj
```

To run the program using 8 cores, use the following command on a *compute node*:

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

Your task for this section is to record in `lab_5_written.txt` 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. You are welcome to use this method in future labs and homeworks.

6 Turning in your lab work and quiz

As in previous labs, you will need to complete a quiz on Coursera and turn in your work before leaving, as follows:

1. Visit rice.coursera.org, select "Fundamentals of Parallel Programming" course, and take the Lab 5 quiz.
2. Check that all the work for today's lab is in the `lab_5` directory. If not, make a copy of any missing files/folders there. It's fine if you include more rather than fewer files — don't worry about cleaning up intermediate/temporary files.
3. Before you leave, create a zip file of your work by changing to the parent directory for `lab_5/` and issuing the following command, "`zip -r lab_5.zip lab_5`".
4. Use the turn-in script to submit the contents of the `lab_5.zip` file as a new `lab_5` directory in your turnin repository as explained in Lab 1. You can always examine the most recent contents of your svn repository by visiting `https://svn.rice.edu/r/comp322/turnin/S13/`*your-netid*.