Lab 4: Real Performance, Work-Sharing and Work-Stealing Schedulers

Instructor: Vivek Sarkar

1 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 -Y ⟨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. Note that 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 Read the SUGAR FAQ

Before going any further, it is important that you familiarize yourself with the SUGAR system by reading the FAQ at `http://rcsg.rice.edu/sugar/faq`. In particular, it is important that you read the following links:

- **Getting Started on SUG@R.** Click on this link and scroll down to the section on "Login Nodes" to understand the difference between login nodes and compute nodes. Make special note of the following comment:

  > "Any user running intensive computational tasks directly on the login node risks disciplinary action up to and including the loss of their access privileges."

- **How do I run graphical applications on a compute node?** Click on this link to see how you might be able to run DrHJ on a SUGAR login node, and still see its GUI on your local computer. Alternatively, you can run DrHJ on your local computer, and transfer files to SUGAR when you need to run them there for performance timings.

3 Compiling and Running HJ programs on SUGAR

This section contains important information for the rest of the lab. Please read through it, and then move on to Section 4.

The simplest way to compile and execute HJ programs on SUGAR is via the command-line interface. To compile an HJ program, Foo.hj, type "*hjc [options] Foo.hj*". The following command-line options are currently available for hjc (type "*hjc -help*" for a summary):

-racedet Enable race-detection when program executes (off by default, only works for basic async and finish constructs)

-dcg Output dynamic computation graph as a dot file when program executes (off by default, only works for basic async and finish constructs)

-rt s Compile program for work-**s**haring runtime (on by default, supports all HJ constructs)

-rt h Compile program for work-stealing runtime with **h**elp-first policy (off by default, only works for basic async and finish constructs)

-rt w Compile program for work-stealing runtime with **w**ork-first policy (off by default, only works for basic async and finish constructs)

-classpath ⟨**path**⟩ Search path for class files

-sourcepath ⟨**path**⟩ Search path for hj source files (must include -classpath if this option is used)

-destdir ⟨**path**⟩ Set the location where output classes from hjc should be placed

-version Print version number of the HJ compiler. Please include this version number when reporting any problems to comp322-staff.

To execute a compiled HJ program, Foo.hj, type "*hj [options] Foo [args]*", where *args* are the command-line arguments for your program'. The following command-line options are currently available for hj (type "hj -help" for a summary):

-places ⟨**p**⟩**:**⟨**w**⟩ Set number of places and workers per places. The default value is $p = 1$ and $w =$ number of processors in the system.

For now, you will be working with 1 place, so you can use *-places 1:n* to run an HJ program with n workers. Since a SUGAR compute node has 8 cores, the best value for n will usually be $n = 8$ (which is the default for SUGAR).

-perf=true Output abstract execution metrics when program executes (off by default, only works for basic async and finish constructs)

-fj Use Fork-Join variant of work-sharing runtime (off by default, assumes that program has been compiled for work-sharing execution)

-version Print version number of the HJ runtime system. Please include this version number when reporting any problems to comp322-staff.

-mx ⟨**size**⟩ set max heap size, e.g., -mx 8000M sets the max heap size to 8GB

-classpath ⟨**path**⟩ Search path for class files

-J⟨**arg**⟩ Pass ⟨arg⟩ directly to Java runtime (use with caution)

To download files on SUGAR, you have two options:

1. Download the file using a web browser on any computer, and then transfer (via sftp or scp) the file to your SUGAR account.
2. Use the *wget* command. If you type the command "*wget URL*" in SUGAR, it will retrieve the file from URL into your local directory e.g.,
   wget http://www.cs.rice.edu/~vs3/downloads/nqueens.hj

Whenever possible, we will try to make copies of files locally available on SUGAR in /users/COMP322.

4 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` example program associated with Lecture 9 in the course web page, or copy it into your local SUGAR directory 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 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.

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

The *async seq* clause was introduced in Lecture 9, and 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

6 Experimenting with the work-stealing scheduler

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

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.

7 Experimenting with futures

In this section, we will experiment with the ArraySum2.hj example introduced in Lecture 7. You can obtain a local copy by typing "*cp -p /users/COMP322/examples/ArraySum2.hj .*" .

Since this program contains futures, it can only be used with the work-sharing scheduler. Try and run this program for an input size of 100 million, and use the *seq* clause to obtain real speedup.

You will need to edit the *seq* clauses and insert timing statements for this exercise.