Prof. Jingke Li (lij@pdx.edu); Classes/Labs: TR 1400-1550 @FAB 150, 88-10; Office Hours: TR 1300-1400 @FAB 120-06

Lab 1: Getting Started

The goal of this lab is to set up the programming environments for the different languages that you'll be using in this class; and to try out a few small programs. For future labs, you may develop your programs on your own computers, but you need to make sure that they run correctly on the intended target system, which will either be the multicore Linux server, `babbage.cs.pdx.edu`, or the CS Linux Lab system, `linuxlab.cs.pdx.edu`.

1 Setup Your Environment

1. Log in to the linux lab system. If you are remote, you may login as follows:

   ```
   yourmachine> slogin <login-name>@linuxlab.cs.pdx.edu
   ```

   (If you are remote login for the first time, you need to answer "yes" to the authentication question.)

2. By default, you should have `bash` shell. You can verify this:

   ```
   linux> echo $0
   /bin/bash
   ```

   If you are not using `bash`, environment variable setup (see later) format will need to be adjusted.

3. Run `addpkg` to add three packages, adaconda (Python 3.7), chapel-1.25, and java8, if you don't already have them:

   ```
   linux> addpkg
   ... (a listing of available packages)
   ```

   Use mouse or arrow and tab keys to select the packages, then select `<OK>` and press `enter`. To effect the selection, you need to logout your session and re-login.

4. Check that you have access to the MPI compiler, `mpicc`:

   ```
   linux> which mpicc
   /usr/bin/mpicc
   ```

   If you don't have access, please let the instructor know.

2 Compile and Run Sample Programs

A set of sample programs using different parallelization tools are provided to you. (A `Makefile` for compiling them and a `runall` script for executing them are also provided.) Follow the instructions below to try them out.

2.1 Pthread (`sum-pthd.c`)

To compile Pthread programs, use `gcc` with `-pthread` flag:

```
linux> gcc -pthread -o sum-pthd sum-pthd.c
```

The compiled programs are run just like regular C programs:

```
linux> ./sum-pthd
The sum is 332833500 (should be 332833500)
```

Note that for this `sum-pthd.c` program, the number of threads is hardwired in the program. In the future, we'll see how to make that adjustable.

Exercise 1 Add a `printf` statement in the `worker()` routine to show the thread id and its work range:

```
linux> ./sum-pthd
Thread 0 worked on [0,100)
Thread 1 worked on [100,200)
...
```

2.2 C++ (sum-oo.cpp)

To compile a C++ program that uses its own thread library, we need to turn on the `-pthread` switch:

```
linux> g++ -pthread -o sum-oo sum-oo.cpp
```

Exercise 2 Make the same change as in the Pthread version, *i.e.* add a `cout` statement in the `worker()` routine to show the thread id and its work range. Verify the output.

2.3 Java (Sum.java)

Compile and run `Sum.java`:

```
linux> javac Sum.java
linux> java Sum
```

Exercise 3 This program is not correct. It's missing synchronization. Run this program multiple times until it produces an incorrect result. Can you explain what execution scenario might have caused this incorrect result?

Hint: It may take *many* runs to produce an incorrect answer. To automate multiple runs, you may want to use a simple shell script (which is provided):

runjavasum

```
#!/bin/bash
for ((i=0;i<$1;i++))
do
    java Sum
done
```

Then you can control the number of runs with a parameter:

```
linux> chmod u+x ./runjavasum  // set execution permission
linux> ./runjavasum 40         // run the program 40 times
```

Exercise 4 Insert a locking statement around the statement `sum += psum`:

```
synchronized (lck) {
  sum += psum;
}
```

The variable `lck` is a synchronization (or monitor) object. In Java, *any* object can be used for this purpose. So we can simply define `lck` as:

```
static Object lck = new Object();
```

Change the program accordingly and re-compile and run.

Why do you think the `static` modifier is needed here? Try removing it and re-run the program. What happens?

Exercise 5 Add a print statement in the `run()` method to print out the thread name and the work range. The thread name can be obtained by calling `Thread.currentThread().getName()`.

2.4 OpenMP (sum-omp.c)

To compile OpenMP programs, use gcc with "-fopenmp" flag:

```
linux> gcc -fopenmp -g -o sum-omp sum-omp.c
```

Again, the compiled programs are run just like regular C programs:

```
linux> ./sum-omp
```

Exercise 6 To see the non-intrusive nature of OpenMP, compile the program without the "-fopenmp" flag, and save it in a different target:

```
linux> gcc -g -o sum-omp0 sum-omp.c
```

The resulting target is a normal sequential code, different from the OpenMP target:

```
linux> wc sum-omp0 sum-omp
 154   152  6384 sum-omp0
  36   199 11696 sum-omp
 190   351 18080 total
```

You may further generate and compare their assembly code:

```
linux> gcc -fopenmp -S sum-omp.c       # generate openmp code
linux> gcc -S -o sum-omp0.s sum-omp.c  # generate sequential code
linux> diff sum-omp.s sum-omp0.s       # compare their contents
```

Exercise 7 Insert a printf statement inside the for loop to print out the current thread id, which can be obtained by calling omp_get_thread_num(). For this to work, you also need to include the OpenMP header file:

```
#include <omp.h>
```

2.5 MPI (sum-mpi.c)

To compile MPI programs, use the command mpicc (which is a gcc wrapper):

```
linux> mpicc -g -o sum-mpi sum-mpi.c
```

Before running MPI programs, you need to setup a host file. Copy linuxhosts to your home directory, and add the following line to your shell startup file, .bash_profile:

.bash_profile

```
# for running MPI programs over multiple hosts
export OMPI_MCA_orte_default_hostfile=~/linuxhosts
```

You also need to add the following line to your .bashrc file:

.bashrc

```
# for fixing a known mpirun messaging bug
export HWLOC_COMPONENTS=-gl
```

Note: You need to reload these two files for the settings to take effect, *e.g.*

```
linux> source ~/.bash_profile ~/.bashrc
```

To run an MPI program, use the command mpirun. Most MPI programs (including sum-mpi.c) are written without specifying the number of processes. This information can be specified with a runtime flag -n:

```
linux> mpirun -n 4 ./sum-mpi    // run 4 copies of the program
```

Exercise 8 Add a printf statement in sum-mpi.c to print out the values of two variables rank (current process id) and size (total number of processes). Compile and run the program to verify that four copies of the code are executed.

2.6 Chapel (`sum1.chpl, sum2.chpl`)

Unlike some of the above cases, where `gcc` handles the compilation, to compile Chapel programs, a separate compiler, `chpl`, is needed:

```
linux> chpl -g -o sum1 sum1.chpl
linux> chpl -g -o sum2 sum2.chpl
```

A Chapel program is run with a mandatory flag `-nl <#locales>` indicating the number of locales (*i.e.* hosts) you'd like to use:

```
linux> ./sum1 -nl 1        // run the program over 1 locale
linux> ./sum2 -nl 4        // run the program over 4 locales
```

However, before you can run Chapel programs over multiple hosts, you need to set the following env variables in your shell startup file:

.bash_profile

```
# for running Chapel programs over multiple hosts
export SSH_CMD=ssh
export CHPL_COMM=gasnet
export GASNET_SPAWNFN=S       # note: no space before and after '='
export GASNET_SSH_SERVERS="bevatron boson ..."  # list of host names
```

The list of host names need to be manually copied from the file `linuxhosts`.

Exercise 9 Add a `writeln` statement inside the `compute()` function in `sum2.chpl` to show where the computation takes place:

```
writeln("i=", i, " is computed on locale ", here.id, ", ", here.name);
```

The two predefined variables, `here.id` and `here.name`, refer to the current locale's id and name, respectively.

Try to run the modified program with a smaller `N` value:

```
linux> ./sum2 --N=10 -nl 4
```

Submission

While labs are not graded, they are required. Everyone needs to submit a lab report. For this lab, you are asked to write a short report (half a page is fine), in plain text, summarize your experience. Specifically, include the following:

1. Confirm that you completed all nine exercises.
2. If you encountered issues, discuss how you resolved them.
3. Briefly discuss your current impressions of these different parallelizing tools.

Name your report `lab1-report.txt`, and upload it through the "File Upload" tab in the "Lab 1" folder. (You need to press the "Start Assignment" button to see the submission options.) The submission deadline is the end of tomorrow (Friday).