CSE 4392 Lab Assignment 1

Due June 20, 2002

Goals:

1. Understanding of elementary pthreads programming.

2. Speed-up and efficiency evaluation of a simple parallel program.

Requirements:

1. Convert the program `radixCount.c` that was sent by email to your `ketchup` account to a parallel pthreads program. A listing is attached.

2. Execute your program on `ketchup` or `mustard` with one thread and then with two threads for sorting 20 nillion keys.

3. Execute your program on one of the Compaq systems via the `student` queue. You should vary the number of threads from 1 to 4 for sorting 40 million keys.

4. Write a brief report discussing the speed-up and efficiency of your program on the Linux and Compaq systems

Getting Started:

1. The provided program does the following:

    a. Input a seed value.
    b. Generate a table of random integer values in the range $0 \ldots 2^{21} - 1$ using `srandom()` and `random()`.
    c. Determine the CPU and elapsed times used by `qsort()` to sort the numbers. The values at subscripts `1000000*i` are printed.
    d. Regenerate the same table as 1.b.
    e. Determine the CPUand elapsed time used by the following LSD radix sort that uses counting sort for each simulated ''digit'' of radix $128=2^7$. When the sort is complete, print out a few values as you did in 1.c.

        1. Sort the input table (`a`) based on the low-order seven bits using counting sort. The output will go to a new table (`b`).
        2. Sort the new table (`b`) based on the middle seven bits using counting sort. The output will go to the original table (`a`).
        3. Sort the original table (`a`) based on the high-order seven bits using counting sort. The output will go to the new table (`b`, the same output table as 1.e.1).

2. `qsort()` is used only to show the advantage of using radix sorting. It is not to be parallelized or to be analyzed for the report.

3. You may simply hardcode a seed value into your program.

4. The same approach may be used for parallelizing 1.e.1, 1.e.2, and 1.e.3. You will only need a single lock, but you will need to check-in to a barrier a number of times.

5. Recall that counting sort is based on the following phases:

    a. Count the number of occurences in the input table for each value in the range of possible values.
    b. Determine the positions that will be used in the output table for each value in the range of possible values.
    c. Rescan the input table to copy each value in the input table to an appropriate position in the output table.

    You will find that partitioning the work (i.e. the input table) in a contiguous fashion to be easier than interleaving. The first and third phases should not need coordination, but 5.b will be more tedious. Unlike the sequential version, which used a single `c` table, your version will have a `c` table for each thread (used for 5.a, 5.b, and 5.c) and a global `c` table (used only for 5.b).

6. Before running your program on a quad-processor Compaq, you should check it on the dual-processor Compaq `rar2001`. You will need Secure Shell to login to `rar2001`, but you may use ordinary ftp to retrieve files from the Linux systems. To run your code on the quad-processor Compaq, you will need a script like the following:

```
[weems@rar2001 ~RADIXCOUNT]$ cat rcStudent.bsub
#!/bin/csh
#BSUB -o rcStudent.out
#BSUB -q student
#BSUB -J radix_count
cc radixCountPT.c barrier.c -o radixCountPT -pthread
./radixCountPT
```

   This script is used to submit the job through the `student` queue, with output to the file `rcStudent.out`, and the job name is `radix_count`. To submit your job, use `bsub<rcStudent.bsub`. The commands `bjobs` and `bqueues` are used to check the queue status. Jobs in the `student` queue are ran each evening from 8:00-8:30.

7. The code in Notes 2 is available on `ketchup/mustard/shiva` in `/home/CODE/NOTES02` and on the server accessed by `rar2001` at `/rnas0/weems/4392_class/NOTES02`.