Programming Assignment 3

Due Monday, March 24
(Note the change in the due date)

For programming assignment 3 you should write a C program that uses Pthreads to implement a sorting algorithm known as sample sort. Input to the program comes from the command line and a file. The command line will consist of three positive ints, a string, and possibly a single char. The ints, are, in order, the number of threads, the sample size s and the list size n. The string is the name of the input file. The optional final command line argument, the char '`n`', will suppress output of the sorted list. The file will consist of a list of n ints (separated by white space). These are the keys to be sorted. Output will be the n ints in sorted order and the elapsed time for the sort (not including I/O). You can assume that $s \leq n$, and both s and n are evenly divisible by the number of threads p.

After getting the command line arguments, the main thread should open the input file, read it into a dynamically allocated block of memory, and fork the other threads. After the sort is complete, the main thread should print the sorted list (unless the '`n`' option was included on the command line) and the elapsed time. Note that the elapsed time for the sort should always be printed — even if the command line option '`n`' is included.

Sample Sort

Sample sort is a generalization of an algorithm known as bucket sort. In bucket sort, we're given two values $a < b$, and the keys are assumed to have a uniform distribution over the interval $[a, b]$. So if we divide $[a, b]$ into m "buckets" of equal size and $h = (b - a)/m$, there should be approximately n/m keys in each of the buckets:

$$[a, a + h), [a + h, a + 2h), \ldots, [a + (m - 1)h, b].$$

In bucket sort we first divide the keys among the buckets and then sort the keys in each bucket.

For example, suppose $a = 1$, $b = 10$, the keys are $9, 5, 8, 1, 4, 6, 3, 7, 2$, and we're supposed to use 3 buckets. Then $h = (10 - 1)/3 = 3$, and the buckets are $[1, 4), [4, 7), [7, 10]$. So 3, 1, and 2 go into the first bucket, 5, 4, and 6 go into the second bucket, and 9, 8, and 7 go into the third bucket.

In sample sort, we don't know the distribution of the keys. So we take a "sample" and use this sample to estimate the distribution. The algorithm then proceeds in the same way as bucket sort. For example, suppose we have m buckets and we randomly choose $s > m$ keys from the n keys. Suppose that after sorting the sample keys are

$$a_0 \leq a_1 \leq \ldots \leq a_s,$$

and we choose the keys in slots that are s/m spaces apart as "splitters." If the splitters are

$$b_1, b_2, \ldots, b_{m-1}$$

then we can use

$$[\min, b_1), [b_1, b_2), \ldots, [b_{m-1}, \max]$$

as our buckets.

For example, suppose our sample consists of the following 12 keys:

$$2, 10, 23, 25, 29, 38, 40, 49, 53, 60, 62, 67.$$

Also suppose we want $m = 4$ buckets. Then the first three keys in the sample should go in the first bucket, the next three into the second bucket, etc. So we can choose as the "splitters"

$$b_1 = (23 + 25)/2 = 24, b_2 = (38 + 40)/2 = 39, b_3 = (53 + 60)/2 = 56.$$

(Note that we're using *integer* division here.)

Parallel Sample Sort

In parallel sample sort, each thread is responsible for the contents of one bucket. So there are p buckets. After the threads are started, each thread chooses a random sample of s/p keys from its assigned keys. In this assignment you should use a block partitioning of the keys. So each thread is assigned a contiguous block of n/p keys from the input list.

You should choose the sample by using the random number generator `random()` and seeding it on each thread with `srandom(my_rank+1)`. In order to ensure that the values generated by random are in "in range," you can take remainders:

```
subscript = my_min_subscript + (random() % local_n);
```

In order to avoid choosing the same key twice, you should keep track of the subscripts previously chosen:

```
do {
   subscript = my_min_subscript + (random() % local_n);
} while (subscript was already chosen);
```

Once the sample has been taken, you need to get the splitters, and to do this you need to sort the s keys in the sample. You can do this using parallel count sort. In parallel count sort, we use a block partition of the sample, and each thread counts the number of keys $\leq$ each of its assigned keys. Ties are broken by comparing subscripts:

```
// Current key = list[i]
// My key = list[j]
if (list[i] < list[j])
   less_than++;
else if (list[i] == list[j] && i < j)
   less_than++;
```

Once the subscripts have been determined, each thread can compute a splitter by taking the maximum sample key assigned to the "preceding" thread and calculating

$$\frac{\text{max on preceding thread} + \text{my min}}{2}.$$

(Note that thread 0 won't compute a splitter.) These should be stored in a shared list.

Suppose that in our earlier example, we have $p = 4$ threads. Then the splitters are calculated as follows:

Thread 0	:	Idle
Thread 2	:	`splitters[0]` $= (23 + 25)/2$
Thread 3	:	`splitters[1]` $= (38 + 40)/2$
Thread 4	:	`splitters[2]` $= (53 + 60)/2$

Now each thread can determine which keys it should get from the other threads.

To start this process, each thread should sort its assigned keys using a fast serial sorting algorithm (e.g., the library quicksort function `qsort`). Then by looking at the splitters and its assigned n/p keys, each thread can determine which thread should get which keys. This will generate a $p \times p$ array of ints. If we call the array `C` then `C[q,r]` is the number of list elements on thread q that should go to thread r.

Several additional arrays can help expedite the determination of the destination of the elements of the input list:

- The prefix sums of the rows of `C`:

  ```
  PSRC[q,r] = C[q,0] + C[q,1] + . . . + C[q,r]
  ```

- The sums of the columns of `C`:

  ```
  CSC[r] = C[0,r] + C[1,r] + . . . + C[p-1,r]
  ```

- The prefix sums of the column sums:

```
PSCSC[r] = CSC[0] + CSC[1] + . . . + CSC[r]
```

If the sorted list will be stored in the array `D`, then each thread can collect "its" elements from the input list into the array `D` by using the following algorithm:

```
my_D_count = CSC[my_rank];
if (my_rank == 0)
   my_first_D = 0;
else
   my_first_D = PSCSC[my_rank-1];

my_D = &D[my_first_D];
index = 0;

// Current thread gets elements from each thread
for (th = 0; th < p; th++) {
   // Offset = offset into list of beginning of
   // elements going to my_rank.  PSRC[th, -1] = 0
   offset = th*local_n + PSRC[th, my_rank-1];
   for (elt = 0; elt < C[th, my_rank]; elt++)
      my_D[index++] = list[offset + elt];
}
```

Now each thread can use a serial sorting algorithm (e.g., `qsort`) on the elements in its block of `D`.

An Example

As an example, suppose that we have four threads, and the keys are initially distributed as follows:

```
Thread 0 > 13 17 12  4 21 23  2
Thread 1 > 28  3 22  6  7 25  8
Thread 2 > 10  5  1 24  9 11 16
Thread 3 > 18 20 27 19 15 26 14
```

(Of course, there is only one global list, but it has been block partitioned among the threads.) Then we might choose the sample to consist of the keys

$$4, 23, 13, 8, 6, 25, 5, 16, 10, 26, 20, 15.$$

When these are sorted, we get

$$4, 5, 6, 8, 10, 13, 15, 16, 20, 23, 25, 26.$$

So our splitters are $7, 14$, and 21. So we'll ultimately assign the keys as follows:

```
Thread 0:       keys < 7
Thread 1:  7 <= keys < 14
Thread 2: 14 <= keys < 21.5
Thread 3: 21 <= keys
```

After local sorts the keys are distributed as follows:

```
Thread 0 >  2  4 12 13 17 21 23
Thread 1 >  3  6  7  8 22 25 28
Thread 2 >  1  5  9 10 11 16 24
Thread 3 > 14 15 18 19 20 26 27
```

So the `C` array should be

```
Thread 0 > 2 2 1 2
Thread 1 > 2 2 0 3
Thread 2 > 2 3 1 1
Thread 3 > 0 0 5 2
```

The prefix sums of the rows of `C` are stored in the array `PSRC`:

```
Thread 0 > 2 4 5 7
Thread 1 > 2 4 4 7
Thread 2 > 2 5 6 7
Thread 3 > 0 0 5 7
```

The column sums and their prefix sums are

```
  CSC:  6  7  7  8
PSCSC:  6 13 20 28
```

Now each thread can use this information to gather the keys that, in the final stage, it will sort. To see how this works consider thread 0. Examining the `C` array, we see it should get 2 of its own elements, 2 elements from Thread 1, 2 elements from thread 2, and no elements from thread 3. So it will get `CSC[0] = 6` elements:

```
Thread 0 >  2  4 | 3  6 | 1  5 |
```

The vertical bars denote where the elements from one thread end and another thread begin. For example, 4 is the last element from Thread 0 and 3 is the first element from Thread 1. Similarly, Threads 1, 2, and 3 will get

```
Thread 1 >  12 13 | 7  8 | 9 10 11 |
Thread 2 >  17 |  | 16 | 14 15 18 19 20
Thread 3 >  21 23 | 22 25 28 | 24 | 26 27
```

Assumptions

You can make the following assumptions:

- $s \leq n$ and both s and n are evenly divisible by p.
- The input values will be correct.
- For synchronization you can use mutexes, semaphores, condition variables and barriers. Note that MacOS X does not support unnamed semaphores or Pthreads barriers. So if you develop your code under MacOS, you won't be able to use these. However, your program will be graded on the CS department Linux systems, and these all support unnamed semaphores and barriers.

Submission

You should submit your source file(s) to your SVN cs625/prog3 directory by 11 am on Monday, March 24, and you should get your hardcopy to me by 2pm.

Grading

1. Correctness will be 75% of your grade. Are the command line arguments handled correctly? Do you read in the file correctly? Do you take the sample correctly? Is your sort of the sample correct? Is your final sort correct?
2. Quality of solution will be 20% of your grade. Are the sampling and the redistribution of the keys implemented efficiently? Is the program well-designed and easy-to-read?
3. Static features such as source format and documentation will be 5% of your grade.
4. Extra credit. There will be a contest among the *correct, well-written and well-documented programs.* The fastest will receive 10 points extra credit. The second fastest will receive 5 points extra credit. The contest will be conducted on the CS server, `chimay`.

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's source code. (This includes source code that you might find on the internet.) If you have any doubt about whether what you'd like to do is legal, you should ask me first.