Lab 2: Parallel Sorting

In this lab, you will use Cilk++ to parallelize a sort function. The write-up for this lab is due on Wednesday, February 24 at 11 am.

Reading

- Section 27.3 of CLRS, provided on the Stellar website.
- Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha, "An experimental analysis of parallel sorting algorithms," *Theory of Computing Systems*, Vol. 31, No. 2, 1998, pp. 135–167. Web link provided on Stellar.
- Any background reading on sorting.

Getting started

A serial implementation of three sort functions can be found in `csort.cilk`. The three sort functions are

- merge sort,
- quick sort, and
- low-order-bit radix sort.

The code, as provided, runs a sort on N rows of data, where each row is a 64-bit key followed by 100 bytes of data. The executable takes N as an optional command-line argument, where N defaults to 10^6. Since the program requires about $112 \cdot N$ megabytes, and each `cagnode` has about 8 GB of RAM, you should be able to go safely up to about 64 million rows without thrashing. You can obtain the code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab2/ lab2
```

We use the standard library `qsort` interface. For more details, you can consult the man pages by typing `man qsort` at the command prompt.

The included `Makefile` builds both a parallel version `csort.par` and its serialization `csort.ser`. As given to you, this code has no `cilk_spawn`'s in it.

Running the program produces output like this:

```
$ ./csort.ser 1000000
system qsort(moverows) time randomdata =  0.476078s     const data =  0.287815s
system qsort(pointers) time randomdata =  0.349245s     const data =  0.158229s
quicksort   (moverows) time randomdata =  0.854006s     const data =  1.180131s
quicksort   (pointers) time randomdata =  0.468307s     const data =  0.357168s
mergesort   (moverows) time randomdata =  0.944632s     const data =  0.865978s
mergesort   (pointers) time randomdata =  0.557010s     const data =  0.297900s
radixsort   (pointers) time randomdata =  0.963987s     const data =  0.349216s
```

Each line of output shows two measurements: one for which the sort key is a random 64-bit number, and one where the sort key is a constant. Lines containing the word "moverows" measure the performance when the sort routine must permute the entire row. In this case, `size == sizeof(struct row)`, is about 108 (or maybe a little more due to alignment requirements). Lines containing the word "pointers" measure the performance when an array of pointers is passed, and the size is `size==sizeof(struct row *)`, which is 8 on a 64-bit machine.

The program measures

- the system `qsort` function ("`system qsort`");
- a quicksort that we wrote ("`quicksort`");
- a merge sort that we wrote ("`mergesort`"); and
- a radix sort that we wrote ("`radixsort`").

Thus we can see that the system `qsort` can sort 10^6 rows in 0.48 s, and it can sort pointers a little faster. None of our code is quite as fast as the system `qsort` function, but we get close. Note that radix sort has a different interface than quicksort and merge sort, since it isn't given a function. Instead, it is given rows so it can see the key.

Your assignment is to implement a parallel sort. If you modify one of the provided codes (recommended), you may wish to delete or comment out the calls to the functions that you aren't working on. For example, for merge sort you might start by writing the parallel merge sort or quicksort that has $\Theta(\log N)$ parallelism by simply spawning the recursive calls. But, we expect you to get more parallelism.

Here are some ideas to get you started:

- For merge sort, consider the algorithm in Section 27.3 of CLRS.
- For quicksort, one way to parallelize the partitioning step is to determine in parallel whether each element is less than, equal to, or greater than the pivot. Given an array $\langle l_0, l_1, l_2, \ldots, l_{m-1} \rangle$, where l_i says that element i is less than the pivot, you can compute the prefix sum

  $$\langle 0, l_0, l_0 + l_1, l_0 + l_1 + l_2, \ldots, l_0 + l_1 + \cdots + l_{m-2} \rangle ,$$

  If $l_i = \text{TRUE}$, then the ith element of the prefix sum tells you the index of where element i should go in the output of the partitioning step. Problem 27-4 of CLRS shows one way to perform a prefix computation using $\Theta(\lg m)$ span and linear work.
- For radix sort, you need to parallelize the construction of the histogram. Some ideas you might try would be to employ an array of reducers for the histogram. Or you might build a per-worker histogram and add up the histograms at the end by hand. You may need to reduce the number of histogram entries to limit the memory pressure. You also might want to consider a high-order-bit radix sort, which has better cache locality.
- If you are ambitious, you can look into other sorting algorithms such as sample sort or shellsort.

Be wary of unconsciously turning this assignment into your term project, however. You should spend only about 12 hours on this lab.

Ideas for a term project

To turn this lab into a term project, you and your partners could parallelize several sorts and compare the various algorithms. Several students could work together to implement even more sorts (Batcher sort, bitonic sort, shellsort, etc.) A paper explaining which sorts are suitable for multicore systems (and why) may be publishable.

We'll also be discussing cache-oblivious sorting algorithms later, which can be its own project.

You could implement an out-of-core sort (in which the data is on disk and is too big to fit in RAM). There is relatively little literature that talks about multicore programming for out-of-core problems.