# **Comp 633: PA 2**

We chose to parallelize quicksort using Cilk for this assignment. We parallelized both the recursive steps, and the partition step.

**Algorithm**:

To parallelize the recursive steps, we simply used the cilk\_spawn function on the first recursive step. It looked something like this:

cilk\_spawn quicksort(left, splitPoint-1);

quicksort(splitPoint+1, right);

cilk\_sync;

Essentially, if a worker thread is available, it will execute the first recursive call, and the current thread will execute the second recursive call. Since quicksort is a divide-and-conquer algorithm, parallelizing this step was straight forward. The more complex part of our algorithm was parallelizing the partition step, which I will explain below:

Assume you have the following array which you wish to partition.

[38, 81, 22, 48, 13, 69, 93, 14, 45, 58, 79, 72]

First, like the normal quicksort algorithm, we must select a pivot. Since we have a randomly generated array, we can simply select the rightmost element as our pivot, which is 72. Now, in parallel, we will reorder our array such that all elements less than or equal to 72 are before it, and all elements greater than 72 are after.

First, we need to classify each element as either “less than or equal to pivot”, or “greater than pivot.” We will use two flag arrays, lte (less than or equal), and gt (greater than) to store this information. Assign n/p values to each processor, and in parallel, place a flag if the value meets the condition for that flag array. For the above array, we will have:

lte: [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1]

gt: [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]

Next, we need to determine which index each element occupies in the partitioned array. To do this, we can do a parallel prefix sum on both lte, and gt.

lte: [1, 1, 2, 3, 4, 5, 5, 6, 7, 8, 8, 9]

gt: [0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3]

Using this, we can iterate through the original array, and for each value, find the proper index using lte and gt. If the value is less than the pivot, its index in the final array is lte[i]. On the other hand, if the value is greater than the pivot, its final index is gt[i]+lte[n-1]. This is demonstrated below:

[38, 81, 22, 48, 13, 69, 93, 14, 45, 58, 79, 72]

lte: [1, 1, 2, 3, 4, 5, 5, 6, 7, 8, 8, 9] gt: [0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3]

[38, 22, 48, 13, 69, 14, 45, 58, 72, 81, 93, 79]

This is a properly partitioned array now.

**Complexity**:

-O(n/p)+O(lgp) for the partition step.

-lg(n) total recursive steps (assuming the pivot divides the array into half each time).

Total: O((n\*lg n)/p)

**Optimizations:**

We added a few optimizations to improve the performance of our program.

1. First, we computed the prefix sum of both lte and gt in one single call. This ensured that we call parallel\_prefixsum() fewer times, and also reduced the overhead of calling cilk\_spawn.
2. Since we generated a random array, we simply chose the rightmost value in the array as the pivot in our partition step. Using a random array, it is unlikely that we will chose a bad pivot consistently if we select the right most value. This saved time because calling rand() is slow.
3. We degraded to a sequential partition once our sub arrays reached to about 70% of our input array. This is because the overhead of calling cilk\_for exceeded the gain of parallelizing the partition after some point. Experimentally, we found 0.7 (or 70%) to be a good factor of the sub array. Lastly, we also degraded to insertion sort once our subarrays reached around size 32. This is because for smaller arrays, insertion sort was faster than quicksort.

**Performance:**

Below, there are two graphs displaying different measures of the performance of our implementation.

**Performance Bottlenecks:**

From the graphs above, you can see that our parallel quicksort implementation was certainly faster than the sequential version. The first graph shows that for all input sizes, increasing the number of processors caused the execution time to decrease. However, the decrease in execution time was not linear. We can see in the second graph that as we increased the number of processors, the amount of speedup we achieved continuously decreased. We have identified a few potential reasons for this:

1. The regions we are attempting to parallelize may not have enough work load. In Cilk, idle worker threads may steal work from other threads. However, if too many steals can incur an overhead. I think it is likely that this may be happening in our implementation. Our parallel regions do not have enough work to keep the threads busy, and hence, they are constantly finishing their work, and stealing work from other threads. The overhead from the stealing may be preventing a linear speedup here.
2. Due to reasons such as the overhead of work stealing, we degrade to sequential partition when we encounter sub arrays that are less that about 70% of our input array. Hence, we are not parallelizing very much in this algorithm; we only attempt to speed up the very large sub-arrays encountered initially. This also contributes to our lack of linear speed up, since we run sequential partition for much of the execution period.