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.
- 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

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:

```bash
$ ./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, \( \text{size} == \text{sizeof(struct row)} \), 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 \( \text{size}==\text{sizeof(struct row *)} \), which is 8 on a 64-bit machine.

The program measures

- the system \( \text{qsort} \) 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 \( \text{qsort} \) 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 \( \text{qsort} \) 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 \( i \)th 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.
1 Parallel Quicksort: Overview

The first step in parallelizing quicksort is to spawn the recursive calls. However, this approach results only in $O(\log N)$ parallelism. In order to improve this bound, we need to parallelize the partitioning step. I implemented several variations of a partitioning algorithm, which relies on prefix-sum computations in order to reorder the elements in the array. This algorithm achieves parallelism $O(N/\log N)$ and is described below.

2 Partitioning Algorithm

**Intuition.** Given a pivot value, the partitioning step needs to reorder the elements in the array, such that all the elements smaller than the pivot precede all the elements greater than the pivot, creating one/several ‘partition’ point(s) which can be used to split the array in two and recursively sort the resulting parts. There can be more than one valid partition point due to elements that are equal to the pivot and it is up to the implementation to decide how these elements are distributed between the two sub-arrays. This distribution is however an important tradeoff (see Tradeoffs section below).

Intuitively, we want to find an appropriate final position for each input element in parallel with the rest. This means, however, that we cannot perform element swapping in place and need to create a copy of the input data. This also means that unless we want to use locking, we cannot share any state among each parallel thread - that is, we cannot keep track of some potentially useful counters - such as, how many elements less than the pivot we have processed so far.

One way to independently figure out the final position of an element $x_i$ is to derive it based on its index $i$ in the input array and some pre-computed values for that index. For example, for each element $x_i < \text{pivot}$, we can pre-compute the sum $s_i = \sum_{j<i, x_j<\text{pivot}} x_j$ and then move the element $x_i$ to position $s_i$, respectively, in the output array. The computation of the $s_i$’s is a prefix-sum computation on an array $b_1, ..., b_n$, where $b_i = 1$ if $x_i < \text{pivot}$ and 0 otherwise, and it’s at the basis of the implemented algorithm. Since all the elements equal to the pivot can be placed on either partition side, knowing the above information is actually enough to partition all the input elements (we can derive the equivalent sum information for elements greater than the pivot by subtracting the known sum for smaller elements at index $i$ from $i$, assuming that at this point either part includes the equal elements). However, due to a significant decrease in performance on constant input values (described in Tradeoffs), the algorithm pre-computes the prefix-sum for both elements less than the pivot and those equal to the pivot. It can then move all the elements equal to the pivot into the middle - this allows us to remove these elements from further recursive calls to quicksort since they are already in their place and ends up speeding up performance on constant keys significantly.

**Implementation.** The algorithm and its parallel version are presented below. For simplicity, several things do not correspond to the actual implementation. In the actual implementation, the temporary arrays are passed as parameters to the function in order to avoid memory allocation slowdown. The output array $y$ is
Algorithm 1 Partition \((x,y,n,pivot)\)

1: let \(l[1..n], q[1..n], l\_prefs[1..n], eq\_prefs[1..n]\) be 4 temporary arrays
2: for \(i = 1\) to \(n\) do
3: if \(x[i] < pivot\) then
4: \(l[i] \leftarrow 1\)
5: \(eq[i] \leftarrow 0\)
6: else if \(x[i] == pivot\) then
7: \(l[i] \leftarrow 0\)
8: \(eq[i] \leftarrow 1\)
9: else
10: \(l[i] \leftarrow 0\)
11: \(eq[i] \leftarrow 0\)
12: end if
13: end for
14: \(prefix\_sum(l, l\_prefs, n)\)
15: \(prefix\_sum(eq, eq\_prefs, n)\)
16: for \(i = 1\) to \(n\) do
17: if \(l[i] == 1\) then
18: \(y[l\_prefs[i]] \leftarrow x[i]\)
19: else if \(eq[i] == pivot\) then
20: \(y[l\_prefs[n] + eq\_prefs[i]] \leftarrow x[i]\)
21: else
22: \(y[l\_prefs[n] + eq\_prefs[n] + i - l\_prefs[i] - eq\_prefs[i] + 1] \leftarrow x[i]\)
23: end if
24: end for

Algorithm 2 Parallel Partition \((x,y,n,pivot)\)

(parallel mem_copy of the input data)

cilk_for (i = 1 to n)
if \(x[i] < pivot\)
\(l[i] = 1\)
\(eq[i] = 0\)
else if \(x[i] == pivot\)
\(l[i] = 0\)
\(eq[i] = 1\)
else
\(l[i] = 0\)
\(eq[i] = 0\)
parallel_prefix_sum(l, l\_prefs, n)
parallel_prefix_sum(eq, eq\_prefs, n)
cilk_for (i = 1 to n)
if \(l[i] == 1\)
y[l\_prefs] = x[i]
else (if eq[i] == pivot)
y[l\_prefs[n] + eq\_prefs[i] ] = x[i]
else
y[l\_prefs[n] + eq\_prefs[n] + i - l\_prefs[i] - eq\_prefs[i] + 1] = x[i]

actually the input array to our sorting function and it needs to be first copied into \(x\). Finally, the partition function actually returns two values: the number of elements below the pivot and the number of elements equal to the pivot, using which we can remove the equal elements from further recursive calls.

In order to parallelize this partition function, we parallelize the two for loops using cilk_for and the computation of the prefix−sum discussed further.

3 Prefix-Sum

The straightforward approach to compute the prefix-sum is to add the previous sum value to the current value in the input array but this can’t be easily parallelized because it requires knowledge of values that may not have been computed yet. To parallelize prefix-sum more efficiently I used two algorithms suggested in Problem 27-4 of CLRS, Algorithm 3-4 analyzed below.
Algorithm 3 prefix-sum-nested \((x, y, i, j)\)

1: if \(i == j\) then
2: \(y[i] \leftarrow x[i]\)
3: else
4: \(k \leftarrow \lfloor (i + j)/2 \rfloor\)
5: \(s\) \(\text{prefix-sum-nested}(x, y, i, k)\)
6: \(s\) \(\text{prefix-sum-nested}(x, y, k+1, j)\)
7: end if
8: \(s\) \(\text{sync}\)
9: for \(l = k+1\) to \(j\) do
10: \(y[l] \leftarrow y[k] + y[l]\)
11: end for

Algorithm 4 prefix-sum-twopass \((x, y, n)\)

1: let \(t[1..n]\) be a temporary array
2: \(y[1] \leftarrow x[1]\)
3: if \(n > 1\) then
4: \(s\) \(\text{scan-up}(x, t, 2, n)\)
5: \(s\) \(\text{scan-down}(x[1], x, t, y, 2, n)\)
6: end if

Algorithm 5 scan-up \((x, t, i, j)\)

if \(i == j\) then
1: return \(x[i]\)
else
2: \(k \leftarrow \lfloor (i + j)/2 \rfloor\)
3: \(t[k] = s\) \(\text{spawn scan-up}(x, t, i, k)\)
4: \(right = \text{scan-up}(x, t, k+1, j)\)
5: \(s\) \(\text{sync}\)
6: return \(t[k] + right\)

Algorithm 6 scan-down \((v, x, t, y, i, j)\)

if \(i == j\) then
1: \(y[i] \leftarrow v + x[i]\)
else
2: \(k \leftarrow \lfloor (i + j)/2 \rfloor\)
3: \(s\) \(\text{scan-down}(v, x, t, y, i, k)\)
4: \(s\) \(\text{scan-down}(v+t[k], x, t, y, k+1, j)\)
5: \(s\) \(\text{sync}\)
6: end if

4 Analysis

1. Recursive Call Spawning

The analysis for the program that just spawns recursive calls and uses Hoare’s partition function is similar to the one done in lecture for Merge sort. The partitioning step takes \(\Theta(n)\) time and the size of each subtree depends on how balanced the partition was. In the worst case scenario, we have a totally unbalanced partition, 1 to \(n - 1\) elements, and in the best case we split exactly in half.

Worst case work: \(T(n) = T(n-1) + \Theta(1) + \Theta(n) = \Theta(n^2)\)

Best case work: \(T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)\)

Avg casework: can be shown to also result in \(\Theta(n \log n)\) (CLRS).

The span will be the time it takes to sort the largest partition plus the time it takes to partition the array:

Avg case span: \(T(n) = T(an/b) + \Theta(n)\) (where \(a < b\) = \( \Theta(n)\)

Therefore, the parallelism we expect is \(\Theta(n \log n/n) = \Theta(\log n)\).

Apart from the prefix-sum work and span, the work and span of each cilk_for loop (including the input copy loop) in Algorithm 2 is \(\Theta(n)\) and \(\Theta(\log n)\) respectively.
2. Nested Parallel Partitioning (Algorithm 3)

Prefix – sum Work: \( T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n) \)
Prefix – sum Span: \( T(n) = T(n/2) + \Theta(\log n) = \Theta((\log n)^2) \) (Case 2)
Prefix – sum Parallelism: \( \Theta(n\log n/(\log n)^2) = \Theta(n/\log n) \).

Algorithm 3 Work: \( T(n) = 3\Theta(n) + \Theta(n\log n) = \Theta(n\log n) \)
Algorithm 3 Span: \( T(n) = 3\Theta(\log n) + \Theta((\log n)^2) = \Theta((\log n)^2) \)
Algorithm 3 Parallelism: \( \Theta(n\log n/n) = \Theta(n/\log n) \).

The total span for quicksort is then \( T(n) = T(n/2) + \Theta((\log n)^2) = \Theta((\log n)^3) \) (CLRS Ex 4.6-2) and so the parallelism is \( \Theta(n\log n/(\log n)^3) = \Theta(n/(\log n)^2) \).

3. Two – Pass Parallel Partitioning (Algorithm 4)

For both scan – up and scan – down, we get the following:
Prefix – sum Work: \( T(n) = 2T(n/2) + \Theta(1) = \Theta(n) \)
Prefix – sum Span: \( T(n) = T(n/2) + \Theta(1) = \Theta(\log n) \)
Prefix – sum Parallelism: \( \Theta(n\log n/n) = \Theta(n/\log n) \).

Algorithm 4 Work: \( T(n) = 4\Theta(n) = \Theta(n) \)
Algorithm 4 Span: \( T(n) = 4\Theta(\log n) = \Theta(\log n) \)
Algorithm 4 Parallelism: \( \Theta(n\log n/n) = \Theta(n/\log n) \).

The total span for quicksort is then \( T(n) = T(n/2) + \Theta(\log n) = \Theta((\log n)^2) \) (Case 2) and so the parallelism is \( \Theta(n\log n/(\log n)^2) = \Theta(n/\log n) \)

5 Performance

When comparing the serial version of the new partition function to other sorts, we can see that it runs slower than most sorts except for the constant data set (this runs fast due to removing the values equal to the pivot from recursive calls). In general, we observe that the performance on pointers data is much better - since we can’t partition in place, we need to copy the input array which will take a longer time when we move the actual rows (Figure1-6).

The increase in parallelism from \( O(\log n) \) to \( O(n/\log n) \) is significant: for \( N=1,000,000 \), we could roughly go from parallelism of 20 to 50,000. If the number of cores on our machines was higher than 20 then this would actually be an upper bound on the speedup we could get. In fact, when we run Cilkview on the code that only spawns recursive calls to quicksort, it reports a parallelism of 7, which actually becomes a limiting factor. (The reason why we might get a parallelism of 7 instead of something slightly higher might be our base case (no recursion for \( n \leq 4 \)) and a constant factor of 1/2 that we ignored in our theoretical analysis of the parallelism).

Running Cilkview on the parallelized partitioning with prefix-sum Algorithm 3 and Algorithm 4, reports a very high parallelism (Table 1). Since the parallelism and burdened parallelism are high, we should expect to see a linear speedup in this case. The speedup looks almost linear for pointers data (Figure 4),
Table 1: Parallel Quicksort: Cilkview Parallelism

<table>
<thead>
<tr>
<th>Input</th>
<th>Alg4</th>
<th>Alg3</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random/Moverows</td>
<td>92367.04</td>
<td>105131.32</td>
<td>7.12</td>
</tr>
<tr>
<td>Constant/Moverows</td>
<td>87027.61</td>
<td>98374.56</td>
<td>8.65</td>
</tr>
<tr>
<td>Random/Pointers</td>
<td>102118.04</td>
<td>109487.62</td>
<td>8.00</td>
</tr>
<tr>
<td>Constant/Pointers</td>
<td>82002.52</td>
<td>96498.50</td>
<td>9.30</td>
</tr>
</tbody>
</table>

however, it flattens around 5 for moverows (Figure 3). This is due to copying the input and my particular implementation of the algorithm, which needs to be optimized for better performance. As we increase N from 1000 - 1000000, we get significantly better speedup results. Algorithm 3 (Figure 5 - 6) runs slower than Algorithm 4 as seen from Analysis (total amount of work is $O(n \log n)$ for Algorithm 3).

Tradeoffs (and Alternatives). We need to be careful what to do with the elements that are equal to the pivot. It is important to keep track of them when the data is not random. In my first implementation I combined the less than and equal data. This ran extremely slowly on constant data. On the other hand, keeping track of the equals requires using more space and an additional prefix-sum computation. One idea would be to randomly decide on which side to put these elements given their position in the array, however, this ends up not being helpful either. Another idea is to use reducers to find how many elements in total are equal to the pivot and less than the pivot and try to come up with a probability function for assigning an equal element to a side. I used two simple oppadd reducers that computed the total number of elements less than and equal to pivot. However, I only played a little with probability parameters and didn’t get better performance in this case.
Figure 1: Recursive - Random/Moverows (rand)

Figure 2: Recursive - Random/Pointers

Figure 3: Alg4 - Random/Moverows (rand)

Figure 4: Alg4 - Random/Pointers
Parallel Radix Sort: In Which More Parallelism is Not Faster Sorting

1 What We Accomplished

We implemented four parallel radix sort algorithms and tested their performance when sorting $10^6$ elements. Parallelizing sorting algorithms is hard, and we were unable to generate an algorithm which ran faster than the C++ standard quicksort algorithm in practice (or even than the standard serial radix sort). However, all-but-one of our parallel radix sort have asymptotically similar behavior to the standard serial radix sort given enough inputs and cores to run on (and enough memory and bandwidth between the cores).

We tackled the problem of parallelizing the LSB radix sort algorithm by focusing on two sections: computing the histogram, and updating the location of element based upon the histograms. Our theoretically most parallel algorithm both computes the histograms and updates element positions in parallel, thereby applying parallelism to the two main section of the radix sort algorithm. We call this algorithm the $Radix_{Parallel-Input-Map}$ algorithm, which was the slowest algorithm of all tested. The three other algorithms all concentrated on providing a parallel computation of the histograms. The $Radix_{Reducer-Array}$ and $Radix_{Reducer-Array-Spaced}$ algorithms both maintain an array of reducer hyperobjects, one for each histogram index (the $Radix_{Reducer-Array-Spaced}$ also takes care to have the reducers be spaced apart in memory to reduce cache consistency issues). Lastly, our $Radix_{Histogram-Merge}$ algorithm splits the input sequence and computes a set of histograms for each subsequence (in parallel), merging the resulting histograms into a single one at the end (in series).

Table 1 shows the timing results of our algorithms on $10^6$ randomly generated elements, as well as the system qsort and the serial radix sort. Table 2 shows results for constant data. It is interesting to note that the serial Radix sort does better with constant data than the Quicksort System. Our $Radix_{Reducer-Array}$ algorithm performed comparably with the serial Radix Serial on constant data.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Num Workers Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort System</td>
<td>0.95 0.78 0.79 0.70 0.70 0.69 0.70 0.70 0.49 0.49 0.48 0.47</td>
</tr>
<tr>
<td>Radix Serial</td>
<td>0.80 0.89 0.74 0.68 0.68 0.68 0.68 0.66 0.66 0.66 0.66</td>
</tr>
<tr>
<td>Radix Reducer-Array</td>
<td>1.60 1.32 1.46 1.14 1.07 1.02 0.95 0.98 0.86 0.83 0.82 0.92</td>
</tr>
<tr>
<td>Radix Histogram-Merge</td>
<td>1.79 1.75 1.71 1.69 1.57 1.51 1.38 1.29 1.43 1.26 1.29 1.81</td>
</tr>
<tr>
<td>Radix Parallel-Input-Map</td>
<td>39 37 35 32 29 27 25 24 20 19 18 19</td>
</tr>
</tbody>
</table>

Figure 1: Timing results for sorting algorithms, random data (in seconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Num Workers Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort System</td>
<td>0.44 0.37 0.42 0.37 0.37 0.37 0.37 0.37 0.25 0.25 0.25 0.24</td>
</tr>
<tr>
<td>Radix Serial</td>
<td>0.18 0.18 0.18 0.18 0.18 0.18 0.18 0.18 0.18 0.16 0.18 0.16</td>
</tr>
<tr>
<td>Radix Reducer-Array</td>
<td>0.19 0.20 0.19 0.19 0.19 0.19 0.19 0.18 0.17 0.18 0.20 0.27</td>
</tr>
<tr>
<td>Radix Reducer-Array-Spaced</td>
<td>3.06 2.59 2.28 2.19 2.15 2.02 1.92 2.14 1.68 1.69 1.71 1.77</td>
</tr>
<tr>
<td>Radix Histogram-Merge</td>
<td>1.40 1.36 1.39 1.43 1.35 1.28 1.16 1.06 1.23 1.20 1.27 1.31</td>
</tr>
<tr>
<td>Radix Parallel-Input-Map</td>
<td>- - - - - - - - - - - -</td>
</tr>
</tbody>
</table>

Figure 2: Timing results for sorting algorithms, constant data (in seconds)
2 The Reducer Array

The simplest parallel radix sort algorithm we implemented is the *Reducer Array* algorithm. It is also the fastest of the parallel algorithms. We create an array of reducer hyperobjects (supporting the ‘+’ operation), one for every bucket in the histogram. Then, we change our histogram computation loop to use a *cilk for* loop rather than a serial loop. It is interesting to note that we had trouble using pointers of reducer objects and the prefix increment operator, therefore we settled on the following reducer syntax:

\[ \text{opadd_reducer} = \text{opadd_reducer} + 1; \]

The simple *Reducer Array* algorithm turns out to be the fastest with the data tested. However, if we look at a work span analysis we see that we did not get much parallelism.

\[
N = \text{number of input elements}
\]

\[
\text{Work b} = \Theta(N \log(N)), \text{the same as with the serial radix sort}.
\]

\[
\text{Span} = \log(N)(\log(N) + \log(N) + N)
\]

Where the first \( \log(N) \) comes from the outer loop of our sort and the fact that our key size is \( \Theta(\log(N)) \). The second \( \log(N) \) comes from the parallel histogram computation (this factor is \( N \) in the serial algorithm), and the rest comes from updating the positions of the elements inside the loop. As such, we get a parallelism:

\[
\frac{b}{\text{Span}} = \Theta(1)
\]

We modified our algorithm slightly to have the reducer objects spaced far apart in memory (where far in this case is in relation to a cache line length). We did not see any significant speed improvements (we actually saw a decrease in speed). It may be that spacing out the reducers also created more true cache contention, far outweighing the benefits of having separate objects on separate cache lines.

3 The Merging Histograms

The *Histogram Merge* algorithm attempts to parallelize the histogram computation without using any reducer hyperobjects. Instead, the algorithm splits the input into subsections and recursively computes a histogram for each section (in parallel). Then, the computed histograms are merged together into a final version. Figure 3 shows the new code which computes the histogram given a set of data and a start and end index (initially 0 and \( N \) respectively). Note that we have two coarsening thresholds, one for a maximum recursion depth and another for a minimum sequence length before we stop recursion and simply compute the histogram using a normal loop.

With correct coarsening values for the recursion (shows in the code), this was still slower than using the reducer array. However, the work is similar to the *Reducer Array* algorithms, culminating in a parallelism of \( \Theta(1) \). It seems like the array of reducer objects is a lighter-weight option for the radix sort histogram computation.

4 The Parallel Index Map

The *Parallel Input Map* algorithm parallelizes both the histogram computation as well as the updating of element positions. We utilize two sets of reducers in order to be able to compute the histogram and position updates in a race-free manner. We start with the array of reducers similar to the *Reducer Array* algorithm. The histogram computation follows exactly like that of *Reducer Array*. Once the histogram is compute, the algorithm then creates a index mapping (one element per input element) and initializes (in parallel) this mapping to the base offset for that element. The base offset is simply the smallest index given the computed histogram that the element could possibly have. The algorithm then enters an inner loop which updates two things: an active set of elements which have yet to reach their correct position and the index map. When there are no more elements in the active set, the loop terminates.
and the index map contains the correct position for each element for the current round of the radix sort. Once we have computed the index map, we can copy the element to their correct locations (in parallel) and go on to the next round.

The tricky part is maintaining the active element set and updating the index map in parallel. The \textit{RadixParallel-Input-Map} algorithm keeps a flag for each input element denoting whether it is part of the active set or not; initially all elements are active. We also keep an array of min-reducer hyperobjects, which collect the minimum element which is part of the active set for each histogram bucket. The algorithm works by increasing the index map entry by 1 for every element of the active set, then removing the minimum elements (per histogram bucket) from the active set, repeat. Figure 4 shows pseudocode for the inner loop of the \textit{RadixParallel-Input-Map} algorithm. Essentially, the algorithm placed one element from each histogram bucket in its correct place every loop. Furthermore, these computation can be done in parallel (hence we use the min-reducer array).

The work span analysis of the \textit{RadixParallel-Input-Map} is a bit tricky. We start with the total work the algorithm performs (note that this is higher than any other algorithm)

\begin{equation}
T_0 = \log(N) \times (H + N + H + N + N + H + H)
\end{equation}

\begin{equation}
= \theta(\log(N)(N + H))
\end{equation}

Where \(N\) = the number of elements in the input, \(H\) = the number of buckets in our histogram (chosen to be \(H = \Theta(\log(N))\)) and \(\bar{H}\) = the maximum histogram count. Equation 2 is the total work done in the inner loop. In the worst case (relating to \(\bar{H}\)), we get:

\begin{equation}
T_{0\text{ worst}} = \Theta(N^2 \log(N))
\end{equation}

For the best case, we get:

\begin{equation}
T_{0\text{ best}} = \Theta(\log(N) \times (N + \frac{N^2}{\log(N)})) = \Theta(N^2)
\end{equation}

For Span, we have:

\begin{equation}
T_\infty = \log(N) \times (\log(H) + \log(N) + H + \log(N) + \log(N) + \log(H) + \bar{H}(\log(H) + \log(N) + \log(H))
\end{equation}

\begin{equation}
= \Theta(\log(N)(\log(N) + \bar{H}\log(N)))
\end{equation}

With worst and best cases:

\begin{equation}
T_{\infty\text{ worst}} = \Theta(N\log^2(N))
\end{equation}

\begin{equation}
T_{\infty\text{ best}} = \Theta(N\log(N))
\end{equation}

Which gives us the following parallelism:

\begin{equation}
\frac{T_{0\text{ worst}}}{T_{\infty\text{ worst}}} = \Theta(N\log(N)) = \frac{T_{0\text{ best}}}{T_{\infty\text{ best}}}
\end{equation}

It is interesting to note that the best case span is \(\Theta(N\log(N))\), which is comparable to a serial sorting algorithm. This means that it is in fact better (asymptotically) to simply run the serial sorting algorithm on 1 core rather than using up lots of core and still having at least one do the same amount of work. While we tried to find a way to parallelize both computations for radix sort, this is the best way we found, which does give the largest parallelism but is not a good algorithm (practically or theoretically) compared to some of the others.
static void calculate_histogram( unsigned int baseshift, size_t histsize, size_t* histogram, struct row **rowsp, int n, size_t start, size_t end, size_t depth )
{
    if( end - start < THRESHOLD || depth >= DEPTH_THRESHOLD )
    {
        for (size_t i=0; i< histsize; i++) histogram[i]=0;
        for (size_t i=start; i<end; i++) {
            size_t idx = (rowsp[i]->key >> baseshift) & (histsize-1);
            histogram[idx]++;
        }
        return;
    }
    // ok, spawn off split histograms then merge them
    size_t *histogramA = (size_t *) malloc(histsize * sizeof(histogramA[0]));
    size_t *histogramB = (size_t *) malloc(histsize * sizeof(histogramB[0]));
    cilk_spawn calculate_histogram( baseshift, histsize, histogramA, rowsp, n, start, end / 2, ++depth );
    calculate_histogram( baseshift, histsize, histogramB, rowsp, n, end / 2, end, ++depth );
    cilk_sync;
    // merge histograms
    cilk_for( size_t i = 0; i < histsize; ++i )
    {
        histogram[i] = histogramA[i] + histogramB[i];
    }
    // clean up
    free (histogramA);
    free (histogramB);
}

Figure 3: RadixHistogram−Merge code snippet
// a set of element indices which are
// still active and participating
ActiveSet = {N}

// mapping between input and correct location of input
// where index_map[i] = histogram[ histogram_bucket of i ]
index_map[ 1 .. N ]

// min element updated from the given histogram bucket
min_index_updated[ 1 .. H ]

while( |ActiveSet| > 0 ):
cilk_for( i = 0 to N ):
    if i in ActiveSet :
        index_map[ i ] += 1
        bucket_idx = histogram_bucket of i
        min_index_updated[ bucket_idx ] = min( min_index_update d[ bucket_idx ], i )

    // remove min indices for each bucket from active set
    ActiveSet = ActiveSet \ { min_index_updated }

Figure 4: RadixParallel–Input–Map Inner Loop Pseudocode
Lab2: Parallelizing MSD Radix Sort with Cilk++

For this lab I explored parallelizing radix sort. I parallelized LSD and MSD radix sort. This write-up focuses on the design and implementation of my parallel version of MSD radix sort, because it provides better performance and larger parallelism than my parallel LSD version.

**Serial algorithm:** Figure 1 presents the serial algorithm for MSD radix sort. The algorithm takes an array of unsorted keys, divides the keys into two arrays based on the most significant bit, recursively divides each array based on the next most significant bit, then finishes the sort by copying the two now sorted arrays into the original array.

**Work:** Figure 2 presents a recursion tree for the MSD radix sort algorithm in Figure 1, where \( n \) is the number of input elements and \( b \) is the number of bits in a key. The total work is given by

\[
T_1(n, b) = \theta(bn)
\]

because the algorithm performs \( n \) amount of computation at each level of the tree and there are a total of \( b \) levels.

**Parallel algorithm:** Figure 3 presents the parallel algorithm for MSD radix sort. The algorithm uses per-core data structures to avoid race conditions that would occur if each core updated the shared \texttt{spliti} and \texttt{split} variables.

**Span and parallelism:** In parallel MSD radix sort the computation at each node in the recursion tree in Figure 2 is the log of its serial value, because each computation is parallelized. The span is calculated by summing the computation along one branch of the tree in figure Figure 2:

\[
T_w(n, b) = \lg n + \lg \frac{n}{2} + \lg \frac{n}{4} + \cdots + \lg \frac{n}{2^{b-1}} = \sum_{i=0}^{b-1} \lg \frac{n}{s^i} \leq \lg n = \theta(\lg n)
\]

and the the parallelism is:

\[
\bar{P} = \frac{\theta(bn)}{\theta(\lg n)}
\]

**Implementation optimizations:** I use several optimizations to improve serial and parallel performance. The most effective were a lock-free block allocator and common prefix reduction. The lock-free block allocator never unmaps memory with the operating systems kernel to avoid the overhead of kernel crossings and of locking operations in the kernel memory management code. Common prefix reduction identifies the bit prefix common to an array of keys. If a common prefix exists, the implementation skips the recursive calls that examine the common bits.

**Experimental results:** Figure 4 presents the Cilkview analysis for sorting 1000000 keys generated by a uniform random distribution. Cilkview predicts perfect linearly speedup for one to 16 workers; however, the measured performance on 16 cores is only about 8 times single core performance. One possible reason is that workers contend for off-chip DRAM bandwidth. As more workers are added, the off-chip DRAM bandwidth stays constant. On the AMD system I ran the benchmarks on, 3 workers \texttt{memcpy}ing into or out of an array held in DRAM are enough to saturate the DRAM controller.

Figure 5 presents the Clikview analysis for sorting 1000000 keys where 500000 are one and 500000 are zero. This benchmark has smaller parallelism than the uniform random key benchmark, because common prefix reduction reduces \( b \) to 1. The MSD parallel radix sort algorithm performs one iteration on the set of keys, splitting the keys into two arrays, then copying them back into the original array. So, in addition to less parallelism than the uniform random key benchmark, scalability will also be more sensitive to off-chip DRAM bandwidth.

**Other ideas:** It would be interesting to investigate the scalability from increasing the the number of bits \( b \) in a key. In theory, MSD radix sort should scale well as \( b \) increases because the parallelism will increase linearly with \( b \). It would also be interesting to determine if off-chip DRAM bandwidth is the scalability bottleneck for MSD parallel radix sort on uniform random keys.
static void radixsort_recur(struct row **rowsp, unsigned int n, unsigned long mod) {
unsigned long spliti[2] = { 0, 0 }; 
struct row **split[2][n];

if (n == 0 || n == 1) 
    return;

for (unsigned int i = 0; i < n; i++) {
    unsigned long idx = (rowsp[i]->key >> mod) & (0x1);
    split[idx][spliti[idx]++] = rowsp[i];
}

if (mod) {
    radixsort_recur(split[0], spliti[0], mod - 1);
    radixsort_recur(split[1], spliti[1], mod - 1);
}

memcpy(rowsp, split[0], spliti[0] * sizeof(*split[0]));
memcpy(&rowsp[spliti[0]], split[1], spliti[1] * sizeof(*split[1]));
}

radixsort(rowsp, nrows, keysize);

Figure 1: The MSD serial radix sort algorithm.

Figure 2: Recursion tree for MSD radix sort where $n$ is the number of input elements and $b$ is the number of bits in a key.
static void pradixsort_recur(struct row **rowsp, unsigned int n,
   unsigned long mod)
{
    unsigned long spliti[2] = { 0, 0 };  
    struct row **split[2][n];

    unsigned long percore_spliti[ncores][2] = { 0, 0 }; 
    struct row **percore_split[ncores][2][n];

    if (n == 0 || n == 1)
        return;

    cilk_for ( unsigned int i = 0; i < n; i++) {
        unsigned long idx = (rowsp[i]->key >> mod) & (0x1);
        percore_split[idx][percore_spliti[idx]++] = rowsp[i];
    }

    for (unsigned int i = 0; i < ncores; i++) {
        combine(split, percore_split[i]);
        combine(spliti, percore_spliti[i]);
    }

    if (mod) {
        cilk_spawn radixsort_recur(split[0], spliti[0], mod - 1) ;
        radixsort_recur(split[1], spliti[1], mod - 1);
    }

    cilk_sync;

    parallel_memcpy(rowsp, split[0], spliti[0] * sizeof(*split[0]));
    parallel_memcpy(rowsp[spliti[0]], split[1], spliti[1] * sizeof(*split[1]));
}

pradixsort(rowsp, nrows, keysize);

Figure 3: The MSD parallel radix sort algorithm.
Figure 4: Cilkview analysis for uniform random keys.

Figure 5: Cilkview analysis for ones and zeros keys.
Lab 2: Samplesort

Ruben Perez

Feb 24, 2010

1 Introduction

For this lab I chose to implement samplesort. This sorting algorithm parallelizes the sorting process by splitting the N elements into P buckets, where P is the number of processors. Each bucket has a pivot value, and all elements in a bucket are less than this value. This allows us to sort each bucket in parallel using another algorithm, like quicksort, and then simply merging the buckets.

2 SampleSort Algorithm

Before discussing implementation details, a more detailed explanation of the algorithm and parameters used is warranted. An observant reader might note that while the overview seems to imply good parallelism, the elements might not be evenly distributed among the buckets. However, sample sort can guarantee this with high probability. The algorithm has each processor sample the N elements in order to get pivot values. Rather than each processor simply choosing an element at random, the processors each choose S elements at random, where S is some sample ratio. We then sort these samples, and choose every Sth value to be a pivot value. By sampling more elements than we need, we can give a mathematical bound on how likely it is that a bucket is overly large. Discussion of bound is in reference 1, at the end of this report, though it does not include a proof. Having chosen our pivots, we then send all elements to the appropriate bucket, where an element goes into bucket 2 if its value is less than that of the pivot, but greater than the pivot of the bucket 1. Since if the pivot value of B1 is less than than the pivot value of B2, we can guarantee that each element is in B1 is less than each element of B2. This allows us to sort B1 and B2 independently and then simply concatenate the two to get a complete sorted list. To summarize:

1. Each processor chooses S samples
2. Sort the samples
3. Choose every Sth value to be a pivot for a bucket
4. Send all elements to the correct bucket
5. Sort each bucket in parallel
6. Concatenate all buckets
3 Implementation

With the explanation of the algorithm in mind, we can now discuss the implementation details. We noted in Lab 1 that using all cores could be sometimes detrimental to performance, so we took care to make the number of processors a defined constant, NUM WORKERS. For the cagnodes, we set this value to 7, as they are 8 core machines. We also made $S$ a defined constant, set initially to 16. By using the cilk for loops, it is relatively easy to split the work amongst the processors, though at times, its unclear whether cilk for’s divide and conquer method is really effective when the intent is to have each thread do some work. This tradeoff will be discussed a bit later on. For now, a cilk for loops is used to have each processor add a random row to an array of sampled rows. Randomness is provided by the standard rand function. This function returns a random number between 1 and RAND MAX, where RAND MAX depends on the compiler. In this case RAND MAX was sufficiently high enough where no further work was needed to guarantee random numbers in the right range. Since we conveniently already have a quick sort function, this is used to sort the samples. To implement buckets, we use a struct that consists of:

- A pointer to a row
- A cilk list append reducer for row
- A startIndex value
- A bucketSize value

The main difficulty that I believe arises in implementing samplesort is getting all elements into their buckets efficiently. Cilk makes this easy, though perhaps not efficient, by providing reducers, which allow multiple threads to safely perform operations on them, in this case adding elements to a list. Thus, this implementation uses a \texttt{hyperobject\langle reducer\_list\_append\langle row\rangle\rangle;} to keep all the rows. An important note is that Cilk seems to require that any struct used in a a reducer be declared extern C++, else template errors appear at compile time. We allocate an array of these bucket structs of size NUM WORKERS, and then set each’s pivot value to be every $S$th sample. The last bucket’s pivot is set to the highest possible value in order to ensure that every element goes into a bucket, while keeping only one loop condition test, the sentinel pattern. We then use a cilk for loop to iterate through all the elements and add them to the appropriate bucket. In interests of a rapid prototype being developed, a simple loop through the the array of buckets is used to determine the correct bucket, rather than the proper binary search. For small numbers of processors this is not as important, though this must be changes to keep scalability. With all elements in their proper bucket, we then iterate through the buckets to get their sizes and appropriate startIndex. The startIndex represents where this bucket would begin the complete arrays, ie bucket 3’s elements belong in indices 5-12 of the returned array. With each bucket struct properly set, we then copy each bucket back to our input array. Since this function is given an array of elements, and expects it to be sorted in place, we must actually copy the elements to a temporary array, and then overwrite the input array. This is done by having the reducers contain that elements, rather than the elements themselves, and having each bucket copy all its elements to the array, starting from its startIndex. This presents to possible sources of inefficiency. One is the
fact that the reducer is queried twice for its array, once in determining the size, and once in the copy procedure. Another is the fact that there is a memory allocation of the size of the input array being performed bit by bit as elements are added to the reducers. I chose to set these aside for now and focusing on the work-span analysis, rather than to spend time doing possibly premature optimizations, but these concerns are important for a true high performance sort. With the elements copied back, each bucket is sorted using quicksort, and the complete array is now sorted. For both the copying procedure, and the quicksort, a cilk for loop is used.

### 4 Analysis

The final cilk for loop raised some interesting questions. The intent of the algorithm is to have each processor perform a quicksort on a bucket, since the buckets are guaranteed to be of roughly the right size. This would be coded in cilk as:

```cilk
for(int i = 0; i < NUM_WORKERS; i++){
    cilk_spawn quicksort ( (void*)(rows + (buckets[i].startIndex * size)) , buckets[i].bucketSize,
}
```

However, most situations would perform better using a cilk for loop, and using its divide and conquer method to evenly distribute work in cases where the loop of the body may contain little work, like so:

```cilk
for(int i = 0; i < NUM_WORKERS; i++){
    quicksort ( (void*)(rows + (buckets[i].startIndex * size)) , buckets[i].bucketSize,
}
```

The question is which is better, or does it even matter since the divide and conquer may do the same as the first loop for sufficiently large workloads in the body of the loop. To test this, I took a look at the parallelism using cilk view and changing just this final loop(all others used cilk for).

```
cilk_for
Parallelism : 1.82
Burdened parallelism : 1.82
Number of spawns/syncs: 2,000,244
```

```
for spawn
Parallelism : 1.87
Burdened parallelism : 1.87
Number of spawns/syncs: 2,000,246
```

The first thing we note is the rather poor parallelism for BOTH implementations. Setting that aside for the moment, it appears that for sufficiently large
work contained in the body, it does not significantly matter. The parallelism and number of spawn syncs is almost exactly the same.

Now the question becomes, why is parallelism so poor. Given that the above indicates that the spawning isn’t a problem, it is somewhat unclear what is proving the main issue. I believe it may simply be the overhead of the memory allocations in the reducers. I don’t know if this would slow down a program’s parallelism (the memory allocations are not multithreaded?), but it is something I would look into. The other parts of the program do not seem to have a high span. Since each thread should have a reasonable large sort to do, parallelizing quicksort should not help parallelism. The sampling procedure is singlethreaded, but only samples $s^*\text{processor elements}$, and sorts them, which is quite small compared to $N$. In terms of performance, without the parallelism, the overhead of the memory allocations and spawns results in a 3x slowdown as compared to the system $qsort$.

Reference:
http://www.eli.sdsu.edu/courses/spring96/cs662/notes/sampleSort/sampleSort.html
What I did in lab2 is to parallelize radix sort using local per-worker storage. Sorry due to the time constraints I did not have time to dive in to depth of the problem, but here is what I did.

In serial radix sort, the program rearranges the order of the entries based on their value on a subsection of their binary representation. The subsection is moving so that at the end the list will be sorted, and the subsection is bounded in length so rearranging can be done with a constant number of buckets. The most time consuming parts in the radix sort is the calculation of the histograms and the rearranging of the entries, both of complexity $O(n)$, where $n$ is the number of entries. To parallelize both, we divide the list into $P$ sections, where $P$ is the number of workers. Then we compute the histograms for each section respectively. Suppose we have a histogram of length $2^k$ in the serial version, then we have $P$ buckets in the parallel version, each of length $2^k$. In serial version, we use the histogram to compute the starting addresses of the $2^k$ buckets. In parallel version, we use the $P$ histograms to compute the starting addresses of the $P2^k$ buckets. This addresses computing is cheap with complexity $P2^k$ and is done in serial. After we have got the starting addresses of the buckets, we can rearrange the entries to their buckets in parallel, where each worker takes care of one of the $P$ sections.

As we are using cilk_for loop with grainsize one to implement the parallelization of histogram computing, the span inherent to the cilk_for loop is $O(\log P)$, where the loop body is again a loop which scans $1/P$ of the total entries. So the total span of the histogram counting is $O(n/P + \log P)$. For the entry rearranging, the parallelization is done with cilk_for loop with grainsize one, and the inherent span is again $O(\log P)$. The loop body also handles $O(n/P)$ entries, resulting in span $O(n/P + \log P)$. Given there are $W/k$ passes, where $W$ is the width of the key representation, the total span is $O(W/k*(n/P + \log P + 2^kP))$, where the $2^kP$ term is from the computing of bucket addresses. When $N >> P$ and $N >> 2^k$, this becomes $O(W/k + P)$. The radix length $k$ has to be carefully chosen. If $k$ is too small, there will be too many passes. If $k$ is too big, then the address computing cost $2^kP$ will be significant.

The core code of the parallel radix sort is pasted here:

```c
cstatic void radixsort_parallel (struct row **rowsp, int n)
// Choose the digit size to be O(log n).
```
// For each digit, make a histogram.
{
    unsigned int mod;
    for (mod=1; (16<<mod)<n; mod++);
    size_t histsize = 1L<<mod;

    // allocate P histograms, each with histsize = 2^k elements.
    size_t P = cilk::current_worker_count();
    size_t* histogram = (size_t*) malloc (histsize * sizeof(histogram[0]) * P);
    size_t** histograms = (size_t**) malloc (sizeof(histograms[0]) * P);

    // allocate the delimiter of the P sections, each with length n/P
    size_t* delim = (size_t*) malloc ((P+1)*sizeof(delim[0]));
    size_t i;
    histograms[0] = histogram;
    delim[0] = 0;
    for (i=1;i<P;i++)
    {
        histograms[i] = histograms[i-1] + histsize;
        delim[i] = n/P*i;
    }
    delim[P] = n;

    // radix sort passes
    struct row **tmp = (struct row **)malloc(n * sizeof(*tmp));
    unsigned int baseshift=0;
    while (baseshift < 8*sizeof(tmp[0]->key)) {
        for (size_t i=0; i<histsize; i++) histogram[i]=0;
        // parallelized histogram counting
        memset(histogram, 0, histsize * sizeof(histogram[0]) * P);
        #pragma cilk_grainsize = 1
        cilk_for (size_t i=0; i<P; i++) { // parallelized using cilk_for with grainsize one
            int w = i;
            // we don’t need to know which worker
            // we are running on, we just need to pick a section and work on it
            size_t idx;
            unsigned int j;
            // works on w-th section
            for (j=delim[w]; j<delim[w+1]; j++)
            {
                idx = (rowsp[j]->key >> baseshift)&(histsize-1);
                histograms[w][idx]++;
            }
        }
    }
}
// compute the bucket starting addresses
size_t sum=0, len_1, len_2;
for (size_t i=0; i<histsize; i++) {
    len_2 = histograms[0][i];
    histograms[0][i]=sum;
    for (size_t j=1; j<P; j++)
    {
        len_1 = histograms[j][i];
        histograms[j][i] = histograms[j-1][i] + len_2;
        len_2 = len_1;
    }
    sum = histograms[P-1][i] + len_2;
}

// parallelized entry rearranging,
// almost the same as parallelized histogram counting
#pragma cilk_grainsize = 1
cilk_for (size_t i=0; i<P; i++) {
    int w = i;
    unsigned int j;
    size_t idx;
    // works on w-th section
    for (j=delim[w]; j<delim[w+1]; j++)
    {
        idx = (rowsp[j]->key >> baseshift)&(histsize-1);
        tmp[histograms[w][idx]++] = rowsp[j];
    }
}
memcpy(rowsp, tmp, n*sizeof(*tmp));
baseshift += mod;
}
free (tmp);
free (histogram);
free (histograms);
free (delim);

The code runs on a million elements with the following performance, which is visualized in Figure 1:

<table>
<thead>
<tr>
<th>worker_count</th>
<th>random data</th>
<th>const data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.940913s</td>
<td>0.381388s</td>
</tr>
<tr>
<td>2</td>
<td>0.521748s</td>
<td>0.242077s</td>
</tr>
<tr>
<td>3</td>
<td>0.413956s</td>
<td>0.224130s</td>
</tr>
<tr>
<td>4</td>
<td>0.364603s</td>
<td>0.207743s</td>
</tr>
<tr>
<td>5</td>
<td>0.322032s</td>
<td>0.195044s</td>
</tr>
<tr>
<td>6</td>
<td>0.299186s</td>
<td>0.177776s</td>
</tr>
<tr>
<td>7</td>
<td>0.291150s</td>
<td>0.184991s</td>
</tr>
</tbody>
</table>
As we can see, it has good speed up with a small worker number, but starts to slow down at 4-5 workers and curve back at 8 workers. To find out the reason, let's first check if it is memory bounded. We compare the running time of one instance of sequential radix sort vs. 4 instances of them.

**One instance:** random data = 16.883121s  const data = 3.817698s

**Four instances:**
- random data = 22.130965s  const data = 5.147498s
- random data = 22.191348s  const data = 8.027179s
- random data = 22.187344s  const data = 8.046829s
- random data = 22.411908s  const data = 6.760992s

We already see some significant slowing down, so the slow down at 4-8 cores is probably due to memory bandwidth.

Parallel radix sort has memory access patterns with low locality. The machine cagnodel has Xeon X5460 processor, which has 12MB cache. The total cache of two processors has 24MB cache. The key alone has 8MB size, with 100MB satellite data. This sorting version has a list of pointers pointing to
the entry data structure, which has key and satellite data, so essentially we are referencing 108MB data in memory. It will be better to make a pointer structure self-containing the key, so it does not need to refer to the 108MB data for sorting. This data structure will have a key and the actual entry address in it, resulting in 16MB data for 1M entries. This should fit in L2 cache and should improve the performance. I’d be very interested to see if it actually speeds it up by containing the key in the pointer structure if given more time...

Also it looks like radix sort is said to be perfect for DNA sorting according to an webpage...
Parallel Sorting Algorithms

1 Merge Sort

<table>
<thead>
<tr>
<th>Variant</th>
<th>pointers</th>
<th>pointers</th>
<th>rows</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>random</td>
<td>const</td>
<td></td>
<td></td>
</tr>
<tr>
<td>serial</td>
<td>10.018781</td>
<td>5.947104</td>
<td>18.707596</td>
<td>17.593525</td>
</tr>
<tr>
<td>naive</td>
<td>3.742065</td>
<td>2.284285</td>
<td>10.020986</td>
<td>9.791809</td>
</tr>
<tr>
<td>basic</td>
<td>3.901268</td>
<td>3.207182</td>
<td>9.718361</td>
<td>9.441949</td>
</tr>
<tr>
<td>using</td>
<td>3.984483</td>
<td>3.215597</td>
<td>9.220462</td>
<td>9.085009</td>
</tr>
<tr>
<td>flipflop</td>
<td>3.793529</td>
<td>3.065596</td>
<td>5.646143</td>
<td>5.457811</td>
</tr>
</tbody>
</table>

Figure 1: Merge sort times sorting 10,000,000 numbers on 8 cores (non-parallel copy).

The timing results for the various merge sort implementations are given in Figure 1. The details of the different variants are as follows:

- **serial** Standard (provided) merge sort with no parallelism.
- **naive** A naive parallelization of the standard merge sort where only the recursive calls are made in parallel.
- **basic** Parallelization of recursive calls and a parallel merge implementation based on the algorithm in CLRS 27.4. Note that using this algorithm, we no long get in place sorting.
- **using** Same as basic, but avoid reallocating the temporary array on each recursive call.
- **flipflop** Optimization of using where we avoid copying if we can and sort one array into another and then do a copy at the top if the data is in the wrong array.

Unsurprisingly, the serial version is the slowest. We see a substantial jump in performance between the serial and naive versions because we are just starting to farm work out onto other processors, even if just for the recursive calls. I expected a slightly larger increase in performance from naive to basic. Even decreasing the size of the payload associated with each key doesn’t speed up basic over naive; in fact, basic actually widens the gap between the two variants with naive still being faster. This is could be due to the implementation of the parallel prefix computation which I wasn’t able to go into in further detail.

The final flip-flop change helps a lot when copying rows and even a little bit when copying pointers. Throughout the entire execution, this piece of code needs to do very few copies, only when two recursive branches differ in their resulting placement, i.e. the flipflop algorithm can choose to put its return value in either the input or the output parameter and it is the caller’s job to handle it appropriately. We only incur the cost of a copy when one of the results ends up in the input buffer and the other ends up in the output buffer.

Scalability results for merge sort are given in Figure 4. Burdened parallelism is near perfect for both of the parallelized implementations. The flipflop algorithm improves the serial performance but doesn’t seem to significantly affect the parallelism.

2 Quick Sort

The timing results for the various quick sort implementations are given in Figure ??

The details of the different variants are as follows:
Scalability results for various implementations of mergesort sorting 5,000,000 elements.

<table>
<thead>
<tr>
<th>Variant</th>
<th>pointers random</th>
<th>pointers const</th>
<th>rows random</th>
<th>rows const</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>9.401280</td>
<td>7.369953</td>
<td>9.861848</td>
<td>12.688648</td>
</tr>
<tr>
<td>naive</td>
<td>3.227354</td>
<td>2.752700</td>
<td>3.965823</td>
<td>5.518392</td>
</tr>
<tr>
<td>flipflop</td>
<td>5.953440</td>
<td>6.700476</td>
<td>10.725774</td>
<td>10.513014</td>
</tr>
<tr>
<td>system</td>
<td>6.303831</td>
<td>3.500197</td>
<td>8.797423</td>
<td>5.879290</td>
</tr>
</tbody>
</table>

Figure 3: Quick sort times sorting 10,000,000 numbers on 8 cores.

**serial** The provided serial quick sort.

**naive** The provided quick sort with recursive calls performed in parallel.

**flipflop** Parallel quick sort including parallelization of the partition step with the flipflop optimization applied.

**system** The built-in system’s `qsort` function.

The performance improvement between the serial and naive parallelization is excellent, but parallelizing the partition actually has a fairly considerable negative affect on performance (at least for sorting 10,000,000 elements). This could be due to the fact that the parallel partition is not in-place, but I would have expected the flipflop variant to address this in the same way that it did for the merge sort. However, note that the number of times that we have to copy is still larger in the quick sort than the merge sort because the two partitions are likely to be different sizes.

Scalability results for quick sort are given in Figure 3. With the parallelized partition, we get near linear burdened speedup with a very high ceiling (not even visible on the graph). The naive quick sort gives considerably less burdened speedup, but actually has comparable timing results. I couldn’t figure out why the trials operate so far below the burdened speedup line in the flipflop variant.

### 3 Radix Sort

I looked at three variants on the radix sort algorithm, the timing results for these algorithms are given in Figure 5. The implementations have the following characteristics:

**serial** The low-order serial radix sort which was given in the lab.

**low** The parallel version of serial. The implementation uses a reducer histogram but the final get_val operation isn’t parallelized.

**high** A high-order radix sort which recursively sorts each of the histogram buckets in parallel. The same reducer histogram is used for this sort.
Amongst the variants there are considerable speedups in the random data case. The difference suggests that the big problem with the implementation is the fact that the histogram reduction is not parallel. You can tell this because the high variant dynamically selects its histogram size on each recursive call to better fit the size of the input array. This means that it takes a deeper recursion, but the histograms at lower levels are considerably smaller than the histograms for low and serial and since they are all doing sequential reduction, this constitutes a considerable improvement in speed.

I would imagine a parallel reduction to perform much better for both of these algorithms. Though I had difficulties implementing the appropriate parallel reducer. At a basic level, it seems that the code should be relatively similar to the code for the hypervector from lab 1.

Scalability results for the two parallel versions of the radix sort are given in Figure 6. The low-order radix sort has pretty poor burdened parallelism, which the high-order one does pretty well over all. The high-order radix sort also has a much higher ceiling (not shown on the graph) because it is able to spawn off the sorting of each bucket in the histogram.
1 Parallelizing Recursive Mergesort Calls

The first step I took to parallelize mergesort was the obvious parallelizing of the two calls to recursively mergesort both halves of the input array. However, both the theoretical analysis of the resulting parallelism and the practical results show that this does not produce a significant improvement in performance.

Since a serial merge function takes $\Theta(n)$ time, the parallelism of this first attempt at parallel mergesort is:

$$M_1(n) = 2M_1(n/2) + \Theta(n) = \Theta(n \log n)$$  \hspace{0.5cm} (1)

$$M_\infty(n) = M_\infty(n/2) + \Theta(n) = \Theta(n)$$  \hspace{0.5cm} (2)

$$M_1(n)/M_\infty(n) = \Theta(\log n)$$  \hspace{0.5cm} (3)

The output obtained from running this initially parallelized mergesort produced the following speedup and parallelism estimates, which show a small ($\sim \Theta(\log n)$) parallelism:

```
cilkview: running with 8 workers
mergesort (moverows) time randomdata = 0.899810s  const data = 0.889608s
mergesort (pointers) time randomdata = 0.405937s  const data = 0.258167s
```

1) Parallelism Profile
   Parallelism : 11.21

2) Speedup Estimate
   2 procs: 1.74 - 2.00
   4 procs: 2.75 - 4.00
   8 procs: 3.88 - 8.00
   16 procs: 4.89 - 11.21
   32 procs: 5.61 - 11.21

mergesort (moverows) const data
1) Parallelism Profile
   Parallelism : 11.25

2) Speedup Estimate
   2 procs: 1.74 - 2.00
   4 procs: 2.75 - 4.00
   8 procs: 3.89 - 8.00
   16 procs: 4.90 - 11.25
   32 procs: 5.63 - 11.25
mergesort (pointers) random data
1) Parallelism Profile
   Parallelism : 11.87
2) Speedup Estimate
   2 procs: 1.75 - 2.00
   4 procs: 2.80 - 4.00
   8 procs: 4.00 - 8.00
   16 procs: 5.08 - 11.87
   32 procs: 5.88 - 11.87

mergesort (pointers) const data
1) Parallelism Profile
   Parallelism : 12.32
2) Speedup Estimate
   2 procs: 1.76 - 2.00
   4 procs: 2.83 - 4.00
   8 procs: 4.07 - 8.00
   16 procs: 5.21 - 12.32
   32 procs: 6.06 - 12.32

2 Parallelizing the Merge Function

Next, I parallelized the merge function using the algorithm described in the CLRS Section 27.3. This algorithm merges two sorted subarrays by using the median of the first subarray as a cutoff point and recursively merging (1) all values less than the median and (2) all values greater than the median. This implementation involves using a $\Theta(lgn)$ binary search on the second input array to find the partitioning point for separating the second array into the two sets of values. My code for this is shown below:

```c
void p_merge(char *dest, char *a, size_t an, char *b, size_t bn, size_t size, compare_fun compar)
{
    char *p1 = a;
    char *r1 = a + an*size - size;
    char *p2 = b;
    char *r2 = b + bn*size - size;
    char *p3 = dest;

    size_t n1 = an;
    size_t n2 = bn;

    if(n1 < n2) // make sure n2 is always smaller than n1
    {
        // exchange p1 with p2
```
char *tempp = p1;
p1 = p2;
p2 = tempp;

// exchange r1 with r2
char *tempr = r1;
r1 = r2;
r2 = tempr;

// exchange n1 with n2
size_t tempn = n1;
n1 = n2;
n2 = tempn;

if(n1 == 0)
{
    return;
}
else if(n1 < 5 || n2 == 0) // coarsening
{
    merge(dest, a, an, b, bn, size, compar, 1);
}
else
{
    char *q1 = int(n1/2)*size + p1;
    char *q2 = binary_search(q1, p2, r2, size, compar);

    char *q3 = p3 + (q1 - p1) + (q2 - p2);
    memcpy(q3, q1, size);

    size_t size_bottom_parta = (q1 - p1)/size;
    size_t size_bottom_partb = (q2 - p2)/size;

    cilk_spawn p_merge(p3, p1, size_bottom_parta, p2, size_bottom_partb,
                        size, compar);
    size_t size_top_parta = (r1 - q1)/size;
    size_t size_top_partb = (r2 - q2+size)/size;

    p_merge(q3+size, q1 + size, size_top_parta, q2, size_top_partb,
            size, compar);
    cilk_sync;
}

The theoretical runtime for this parallel merge is $PM(n) = PM(3n/4) + \Theta(lgn) = \Theta(lg^2n)$, since the median of one array being used as the partition value recursively reduces the subproblem by at
least 3/4. In order to improve the performance of this parallel merge function, I coarsened it using a threshold value of 4, which produced the smallest execution time of the mergesort algorithm in practice.

Additionally, I modified the initial merge function to use a parallelized memcpy once the merge function got down to the base case of merging an array with a zero-length array.

### 3 Integrating the Merge Function into the Parallel Mergesort

Combining the parallelized merge with the initial parallel mergesort yields a theoretical parallelism of

\[
M(n) = M(\alpha n) + PM((1 - \alpha)n) + O(lgn) = \Theta(n/lg^2n)
\]

where \(1/4 \leq \alpha \leq 3/4\), since each parallel merge may separate the array into two subarrays of size anywhere between \(n/4\) to \(3n/4\).

Originally, I integrated the parallel merge into the mergesort in the manner outlined in the CLRS pseudocode for the P-MERGE-SORT(A, p, r, B, s) function (see pg. 803). However, there were several inefficiencies in such an implementation.

Firstly, the cost of using a parallel mergesort down to an array size of 1 was impractically large. To avoid this, I coarsened the code using the basesort sorting algorithm as the fallback algorithm, in the same manner that the original mergesort did this. For this, a threshold value of 4 again seemed to produce the shortest runtimes in practice.

Secondly, there was sizable overhead associated with allocating a new temporary array for each recursive mergesort call. To solve this, I passed two temporary arrays into the mergesort function, so that each successive mergesort call could reuse the provided temporary arrays to transfer its data from the input array \(\Rightarrow\) temporary array \(\Rightarrow\) output array without race conditions. Since for each recursive call the temporary array at one level becomes the output array for the recursive level below it, two temporary arrays were necessary to prevent any mergesort function from attempting to read from and write to the same temporary storage place. This modification improved the speedup of the mergesort algorithm as well.

The figures below show the speedup for three versions of mergesort: (1) the original parallelized mergesort as outlined in the CLRS (2) the original parallelized mergesort from CLRS, using a parallelized memcpy for the base case merge and (3) the final optimized parallel mergesort, using a parallelized memcpy and with reused temporary array storage space (as described in the paragraph above). The original parallelized mergesort produced the following output:

```plaintext
mergesort (moverows) randomdata
1) Parallelism Profile
   Parallelism : 44059.13
2) Speedup Estimate
   2 procs: 2.00 - 2.00
   4 procs: 4.00 - 4.00
```
Figure 1: Parallelized MergeSort: based on CLRS P-MERGE-SORT algorithm

Figure 2: Parallelized MergeSort: Optimized to use Parallel Memcpy

Figure 3: Parallelized MergeSort: Optimized for Temporary Storage Space

8 procs: 8.00 - 8.00
16 procs: 15.99 - 16.00
32 procs: 31.96 - 32.00
mergesort (moverows) const data
1) Parallelism Profile
   Parallelism : 116.92
2) Speedup Estimate
mergesort (pointers) random data
1) Parallelism Profile
   Parallelism : 26765.31
2) Speedup Estimate
   2 procs: 1.97 - 2.00
   4 procs: 3.83 - 4.00
   8 procs: 7.26 - 8.00
   16 procs: 13.14 - 16.00
   32 procs: 22.06 - 32.00
mergesort (pointers) const data
1) Parallelism Profile
   Parallelism : 184.91
2) Speedup Estimate
   2 procs: 1.98 - 2.00
   4 procs: 3.89 - 4.00
   8 procs: 7.52 - 8.00
   16 procs: 14.06 - 16.00
   32 procs: 24.90 - 32.00

Below is my code for the final parallel mergesort:

```c
void mergesort (void *base, char *tmp, char *tmp2, char *output, size_t n_rows,
               size_t size, compare_fun compar)
{
    if (n_rows<basesort_size) {
        basesort(base, n_rows, size, compar);
        memcpy(output, (char*)base, size*n_rows);
    } else {

        int mid = n_rows/2;
        cilk_spawn mergesort(base, tmp2, tmp, tmp, mid, size, compar);
        mergesort((char*)base+mid*size, tmp2+mid*size, tmp+mid*size,
                  tmp+mid*size, n_rows-mid, size, compar);
        cilk_sync;

        p_merge(output, tmp, mid, ((char*)tmp+mid*size), n_rows-mid, size, compar);
    }
}
```
This final version of the parallelized mergesort produced the output shown below. Comparing this output to the one from before, as well as the speedup graphs for the three versions of mergesort, it can be seen that optimizations made to the original CLRS parallelized mergesort improved both the speedup and the parallelism of the resulting mergesort algorithm.

mergesort (moverows) randomdata
1) Parallelism Profile
   Parallelism : 48975.25
2) Speedup Estimate
   2 procs: 2.00 - 2.00
   4 procs: 4.00 - 4.00
   8 procs: 8.00 - 8.00
   16 procs: 15.99 - 16.00
   32 procs: 31.97 - 32.00

mergesort (moverows) constdata
1) Parallelism Profile
   Parallelism : 650.14
2) Speedup Estimate
   2 procs: 1.99 - 2.00
   4 procs: 3.97 - 4.00
   8 procs: 7.86 - 8.00
   16 procs: 15.40 - 16.00
   32 procs: 29.60 - 32.00

mergesort (pointers) randomdata
1) Parallelism Profile
   Parallelism : 37186.79
2) Speedup Estimate
   2 procs: 2.00 - 2.00
   4 procs: 4.00 - 4.00
   8 procs: 8.00 - 8.00
   16 procs: 15.99 - 16.00
   32 procs: 31.95 - 32.00

mergesort (pointers) constdata
1) Parallelism Profile
   Parallelism : 5194.10
2) Speedup Estimate
   2 procs: 2.00 - 2.00
   4 procs: 4.00 - 4.00
   8 procs: 7.98 - 8.00
   16 procs: 15.92 - 16.00
   32 procs: 31.68 - 32.00
1 Parallel Radix using Reducer Histogram

I first attempted to parallelize Radix sort by using reducers. To do this, I create an array of list append reducers where there is a list reducer for each possible digit in the histogram. Then, pointers are added in parallel to these lists and finally the array is looped over and the values are stored back into `rowsp`.

This initially ran extremely slow. I suspected that this may be improved significantly with smaller histogram sizes. I retested with histogram sizes of 256 and 128, but there was very little improvement. The main problem with this approach is during the concatenation loop. The values should be computed in parallel, but my implementation does not do this. However, a custom reducer for the histogram may also work better since the total space required is known beforehand, so perhaps vectors would be more efficient than lists.

2 Parallelizing using Per-Worker Histograms

After being dissatisfied with the performance of the reducer approach, I implemented a radix sort where I used 8 histograms, and spawned 8 workers that filled their histograms independently. Then, these histograms were combined in parallel (see pseudocode in Section 2.1) This drastically sped up the most time expensive loop in the counting sort, enough so that the last loop in the counting sort became the bottleneck.

Initial tests showed that this ran about 33% faster than the serial radix sort. This speedup is good, however, any additional speedup for this approach would need to come from the final loop in the counting sort. Here, the histogram array is incremented to keep track of where pointers need to be moved to. If I use locks to resolve the race here, stable sorting would break. I would be forced instead to use a reducer. A reducer here would be similar to the histogram reducer implemented before, so that’s an idea I decided not to explore.

2.1 Pseudo-Cilk++

```cilk
void radix_splitmerge(...){
  ... INITIALIZATION ...

  while (baseshift < 8*sizeof(tmp[0]->key)) {

    cilk_for (int i=0; i<s; i++) {
      cilk_for (size_t j=0; j<histsize; j++){
        histotmp[i][j] = 0;
      }
      for(int j = part * i; j < min(part * (i+1), n); j++){
        size_t idx = (rowsp[j]->key >> baseshift)&(histsize-1);
        histotmp[i][idx]++;
      }
    }
  }
}
```
cilk_for (size_t i = 0; i < histsize; i++){  
    histogram[i] = histotmp[0][i];  
    for(int j = 1; j < s; j++)  
        histogram[i] += histotmp[j][i];  
}

... NORMAL REMAINING COUNTING SORT ...

2.2 Parallelism

In each counting sort iteration, I have a serial loop that takes \( \Theta(n) \) time. So the span is \( \Theta(n) \) for each counting sort. The work in each counting sort is also \( \Theta(n) \), so the theoretical parallelism is just 1. However, there is actual parallelism for parts of the counting sort and so there is some speedup (see the cilkview outputs in Figure 1).

![Figure 1: Cilkview graph for radixsort_splitmerge.](image)

3 Most Significant Digit Radix Sort

Determined to investigate alternatives for parallelizing the final loop in radix sort, I implemented radix sort using a most significant digit first approach. To do this, after the first digit has been sorted,
buckets formed by that pass can each be sorted independently without stable sorting. This allows me to spawn different workers for each bucket in the histogram and it also allows me to run the last loop of the counting sort in parallel (see the pseudocode below.) Of course, this method separates the list way too much if the histogram is too large. Because of this, I fixed the histogram size at 256. This breaks the theoretical runtime of radix sort, but sped up the sort in experiments.

3.1 Pseudo-Cilk++ for MSD Radix

```cpp
void radix_msd(...){
    if(n <= 1)
        return;

    ...

    BUILD_HISTOGRAM_NORMALLY()
    ...

    // Use a cilk_for if I can, otherwise, serially...
    if(!(MULTI_BUCKET))
        cilk_for (int i=0; i<n; i++) {
            size_t idx = (rowsp[i]->key >> baseshift)&(histsize-1);
            histlck[idx].lock();
            tmp[histogram[idx]++] = rowsp[i];
            histlck[idx].unlock();
        }
    else
        for (int i=0; i<n; i++) {
            size_t idx = (rowsp[i]->key >> baseshift)&(histsize-1);
            tmp[histogram[idx]++] = rowsp[i];
        }

    memcpy(rowsp, tmp, n*sizeof(*tmp));
    baseshift -= mod;

    if(baseshift <= 0)
        return;

    if(n <= COARSEN_THRESH){
        radixsort_msd(rowsp, n, MULTI_BUCKET=true);
    }else{
        cilk_spawn radixsort_msd(rowsp, histogram[0], MULTI_BUCKET=false);
        cilk_for (size_t i = 1; i < histsize; i++){
            radixsort_msd_help(rowsp + histogram[i - 1],
```
3.2 Coarsening

I decided that coarsening would be very necessary for this algorithm, since the list could be broken down into lists of size one, which would certainly not be a very efficient use of counting sort. My first guess for a coarsening threshold was $10^5$. I tested other thresholds an order of magnitude larger or smaller.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Input Size</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10K</td>
<td>10K</td>
<td>0.004</td>
</tr>
<tr>
<td>10K</td>
<td>100K</td>
<td>0.147</td>
</tr>
<tr>
<td>10K</td>
<td>1M</td>
<td>0.172</td>
</tr>
<tr>
<td>100K</td>
<td>10K</td>
<td>0.004</td>
</tr>
<tr>
<td>100K</td>
<td>100K</td>
<td>0.080</td>
</tr>
<tr>
<td>100K</td>
<td>1M</td>
<td>0.141</td>
</tr>
<tr>
<td>100K</td>
<td>10M</td>
<td>4.472</td>
</tr>
<tr>
<td>1M</td>
<td>10K</td>
<td>0.004</td>
</tr>
<tr>
<td>1M</td>
<td>100K</td>
<td>0.081</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>0.170</td>
</tr>
<tr>
<td>1M</td>
<td>10M</td>
<td>4.486</td>
</tr>
</tbody>
</table>

From these results I concluded that $10^5$ was an acceptable threshold. $10^6$ performs similarly for really large inputs, but does not perform nearly as well around the default input size.

3.3 Histogram Size

After adjusting the histogram size to 512, this algorithm splits into 512 recursive steps which each split again for reaching the coarsening level. This creates an excessive memory allocation overhead. For this reason, I kept the histogram size at 256. However, another approach would be to better control how many recursive calls were made by combining buckets. This would force child calls to be stable, but would likely result in more parallelism in the long run and less work.

3.4 Parallelism

The parallelism of the algorithm when the digit is $\Theta(\log(n))$ is $\Theta(\log(n))$. The span is from the first counting sort to the last, each iteration is half the size of the previous, which sums $\Theta(n)$. The total work is $\Theta(n \log(n))$, so $W/S = \Theta(\log(n))$. However, if the digit size is fixed (as in my implementation)
parallelism decreases with $n$. The parallel calls after each counting sort are related to the digit size, but the first counting sort size increases with $n$. This leads to a situation where the first counting sort becomes more and more burdensome (see Figures 2 and 3.) This is not a problem for my experiments on the 8 core machines, but for larger problems and more cores I would want to explore the alternative version I discussed in the previous section.

2 procs: 1.86 - 2.00  
4 procs: 3.28 - 4.00  
8 procs: 5.30 - 8.00  
16 procs: 7.66 - 16.00  
32 procs: 9.85 - 23.42

Figure 2: Cilkview expected parallelism for \texttt{radix.msd} with $n = 10^6$

2 procs: 1.74 - 2.00  
4 procs: 2.74 - 4.00  
8 procs: 3.87 - 8.00  
16 procs: 4.87 - 11.14  
32 procs: 5.59 - 11.14

Figure 3: Cilkview expected parallelism for \texttt{radix.msd} with $n = 10^7$

4 Performance Comparisons

Of the three implemented parallel radix sorts, \texttt{radix.msd} and \texttt{radix.splitmerge} both were faster than the serial version. \texttt{radix.reduce} did not fare as well. The runtimes of the various versions on an 8 core cagnode machine are shown in Figure 4. As you can see, \texttt{radix.msd} wins by a factor of about 4 over \texttt{radix.splitmerge} on large inputs and by a factor of about 6 over the normal serial radix sort.
Figure 4: Runtimes for the different implementations of radix sort on a log-log plot.
For this lab, I decided to parallelize radix sort. I did this in two ways: (i) locks and cilk_for, and (ii) splitting the histogram between worker threads. I expected the first solution to be easier to implement, but give pretty bad performance; and the second to be more difficult to implement, but give actual parallelization gains.

1 Lock Parallelization

I first parallelized using locks around each histogram element. This was an attempt to get some fine-grain locking by not locking the entire data structure, but even then there is overhead incurred by locking and some contention. As a result, this did not achieve good speed-up.

The code was changed as follows: (i) add locks around each access to the histogram in two places, and (ii) use parallel_copy I implemented. I chose to let the loops over the histogram itself remain serial, as they shouldn't take long ($\text{histsize} \ll \text{n}$).

Code listing:

```c
static void radixsort_parallel_locks (struct row **rowsp, int n)
// Choose the digit size to be $O(\log n)$.
// For each digit, make a histogram.
{
    unsigned int mod;
    for (mod=1; (16<<mod)<n; mod++);
    size_t histsize = 1L<<mod;
    size_t *histogram = (size_t *)malloc(histsize * sizeof(histogram[0]));
    struct row **tmp = (struct row **)malloc(n * sizeof(*tmp));
    lock *locks = new lock[histsize];

    unsigned int baseshift=0;
    while (baseshift < 8*sizeof(tmp[0]->key)) {
for (size_t i=0; i<histsize; i++) histogram[i]=0;
    cilk_for (int i=0; i<n; i++) {
        size_t idx = (rowsp[i]->key >> baseshift)&(histsize-1);
        locks[idx].m.lock();
```

1
Finally, note that lock is simply a cilk::mutex with some char padding to avoid false sharing.

2 Split Histogram Parallelization

There are two steps in this version that must be parallelized: the counting of the matrix, and the copying/sorting of the array to follow the counts in the histogram.

Parallelizing the counting was fairly easy. I used divide-and-conquer over the input and built a histogram for each division. At each step, these histograms were merged, giving a final histogram with the correct counts at the top level.

Code is below:

```c
static void fill_histogram(size_t * histogram, unsigned int baseshift, size_t histsize, struct row ** rowsp, int min, int max)
{
    #define THRESHOLD 65536
    if (max - min < THRESHOLD)
    {
        for (int i=min; i<max; i++) {
            size_t idx = (rowsp[i]->key >> baseshift)&(histsize-1);
            histogram[idx]++;
        }
    }
    size_t sum=0;
    for (size_t i=0; i<histsize; i++) {
        size_t prevsum=sum;
        sum+=histogram[i];
        histogram[i]=prevsum;
    }
    for (int i=0; i<n; i++) {
        size_t idx = (rowsp[i]->key >> baseshift)&(histsize-1);
        locks[idx].m.lock();
        tmp[histogram[idx]++] = rowsp[i];
        locks[idx].m.unlock();
        parallel_copy(rowsp, tmp, n*sizeof(*tmp));
        baseshift += mod;
        free (tmp);
        free (histogram);
        delete [] locks;
    }
}
```
// allocate sub histogram and break problem in half
size_t *histogram2 = (size_t *)malloc(histsize * sizeof(histogram[0]));
for (unsigned int i = 0; i < histsize; i++)
  histogram2[i] = 0;

int middle = ((max - min) / 2) + min;
cilk_spawn fill_histogram(histogram, baseshift, histsize, rowsp, min, middle);
fill_histogram(histogram2, baseshift, histsize, rowsp, middle, max);
cilk_sync;

// merge
for (unsigned int i = 0; i < histsize; i++)
  histogram[i] += histogram2[i];
free(histogram2);

The base case is simply what was provided in the basic radix sort code. In
the recursion, I allocate a new histogram for the spawned call to use and then
recurse. Finally, I “merge” the two histograms by adding them together. As in
the prior case, I did this serially because it was a relatively small step.

The next step, copying/sorting, was considerably more challenging. I chos
the same basic strategy for parallelization – divide and conquer over the input.
Each section will put the values in the correct order for its segment of the input
using a hypervector. I allocate one hypervector per histogram bucket so that
they can be joined correctly. These hypervectors are finally “joined” into the
tmp array that is expected to hold the output (equivalent to get_vec in Lab
1C).

First, the hypervector itself:

struct hypervec
{
  static const int VECTOR_SIZE = 65536;
  struct row * v[VECTOR_SIZE];
  size_t idx;
  hypervec * next;
  hypervec * tail;

  hypervec()
  {
    idx = 0;
  }
next = NULL;
tail = this;
}

~hypervec()
{
    delete next; next = NULL;
tail = NULL;
}

void push_back(row * e)
{
    v[idx++] = e;
}

void append(hypervec * h)
{
    tail->next = h;
tail = h->tail;
}

void join(row ** out, __attribute((__unused__)) size_t expected)
{
    hypervec * p = this;
    size_t offset = 0;

    while (p)
    {
        if (p->idx != 0)
        {
            cilk_spawn parallel_copy<row *>(out + offset, p->v, p->idx);
        }
        offset += p->idx;
p = p->next;
    }
}

This is a fairly straightforward implementation of the hypervector, very similar to what we did in Lab 1C. The only things to note are:

- I had to use a static-sized array instead of a *std::vector in order to get any kind of performance. This is OK because I coarsen on the granularity of the size of this vector, so it will not be overwritten.
- I used the tail pointer idea discussed in class to make appends more effi-
cient.

- I spawn off instances of a parallel copy operation to elegantly handle the case of many, small vectors and a few, large vectors. This should be rewritten to use \texttt{cilk_for}, and I initially had an implementation that did this, but I found for this input that the above implementation had better performance.

The next component is the copying routing that uses the hypervector:

```c
static void copy_histogram_2(hypervec ** h, unsigned int baseshift,
   size_t histsize, struct row ** rowsp,
   int min, int max)
{
    if (max - min < hypervector::VECTOR_SIZE)
    {
        for (int i=min; i<max; i++) {
            size_t idx = (rowsp[i]->key >> baseshift) & (histsize-1);
            h[idx]->push_back(rowsp[i]);
        }
    }
    else
    {
        hypervec ** h2 = new hypervec*[histsize];
        for (size_t i = 0; i < histsize; i++)
            h2[i] = new hypervec;

        int middle = ((max - min) / 2) + min;
        cilk_spawn copy_histogram_2(h, baseshift, histsize, rowsp, min, middle);
        copy_histogram_2(h2, baseshift, histsize, rowsp, middle, max);
        cilk_sync;

        // merge
        for (size_t i = 0; i < histsize; i++)
            h[i]->append(h2[i]);

        delete [] h2;
    }
}
```

This code follows a similar pattern to \texttt{fill_histogram} discussed above – divide and conquer on the input. Again, I used essentially the code provided for the base case (except I use \texttt{push_back}). The only potentially interesting thing is the spawning of the parallel jobs, and the resulting hypervectors are appended.

There is a top-level copying function that calls \texttt{copy_histogram_2} so that the hypervectors are joined after it returns, it is very simple:
static void copy_histogram(struct row ** tmp, size_t * histogram,
 size_t * offsets, unsigned int baseshift,
 size_t histsize, struct row ** rowsp,
 int min, int max)
{
    hypervec ** h = new hypervec*[histsize];
    for (size_t i = 0; i < histsize; i++)
        h[i] = new hypervec;
    copy_histogram_2(h, baseshift, histsize, rowsp, min, max);
    cilk_for (size_t i = 0; i < histsize; i++)
        h[i]->join(tmp + offsets[i], histogram[i]);
    for (size_t i = 0; i < histsize; i++)
        delete h[i];
    delete [] h;
}

The last piece is the sorting function itself, which is nearly identical except
for using the above functions in the right places. We also do a parallel copy.

static void radixsort_parallel_multihistogram (struct row **rowsp, int n)
// Choose the digit size to be 0(log n).
// For each digit, make a histogram.
{
    unsigned int mod;
    for (mod=1; (16<<mod)<n; mod++);
    //printf("n=%d digit size = %d (%ld) (%d passes)\n", n, mod, 1L<<mod, (64+mod-1)/mod);
    size_t histsize = 1L<<mod;
    size_t *histogram = (size_t *)malloc(histsize * sizeof(histogram[0]));
    size_t *offsets = (size_t *)malloc(histsize * sizeof(offsets[0]));
    struct row **tmp = (struct row **)malloc(n * sizeof(*tmp));
    unsigned int baseshift=0;
    while (baseshift < 8*sizeof(tmp[0]->key)) {
        for (size_t i=0; i<histsize; i++) histogram[i]=0;
        fill_histogram(histogram, baseshift, histsize, rowsp, 0, n);
        size_t sum=0;
        for (size_t i=0; i<histsize; i++) {
            size_t prevsum=sum;
            sum+=histogram[i];
            offsets[i]=prevsum;
        }
        copy_histogram(tmp, histogram, offsets, baseshift, histsize, rowsp, 0, n);
        parallel_copy<struct row *>(rowsp, tmp, n);
        baseshift += mod;
free (tmp);
free (offsets);
free (histogram);
}

3 Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random</td>
<td>Const</td>
</tr>
<tr>
<td>Basic radix sort</td>
<td>0.97</td>
<td>0.46</td>
</tr>
<tr>
<td>Locks</td>
<td>2.22</td>
<td>1.07</td>
</tr>
<tr>
<td>Multi-histogram</td>
<td>0.98</td>
<td>0.47</td>
</tr>
</tbody>
</table>

Table 1: Results for radix sort implementations.

Results are given in Table 1. My results are pretty much as expected, although the “multi-histogram” (reducer) version didn’t get as good performance as I hoped. The lock version experienced significant slowdown, and actually got $3 \times$ slower with constant data in the parallel case. (This is clearly due to a single lock serializing the computation.) The reducer version sped up slightly with 8 processors, but not nearly as much as I had hoped.

Interestingly, I ran the reducer version with just the fill_histogram step parallelized and got nearly identical results to those shown. Clearly, whatever parallelization savings I’m getting are from this part. I need to debug where the performance is going in the reducer part of the code.

4 Thoughts

Here are a few other thoughts from my experience with this lab:

- I think mallocs are costing me a lot of time. I could use a memory pool for this and possibly save a lot.

- I did not have enough time to fiddle with my coarsening parameters, etc. I not sure how much of an impact this could make, but what is currently in there (particularly the reducer vector size and the bucket size) are essentially arbitrary to my knowledge.

- I originally wrote the reducer using stl::vector, and it was $4 \times$ slower. That made it by far the slowest version. I realized from debugging that just using the fill_histogram version gave performance gains, so I knew it wasn’t fundamentally parallelization overhead. Making a few minimal changes to use statically allocated chunks gave me all the performance back.
• Right now, I run all of the loops over \texttt{histsize} serially. This is potentially very expensive, as \texttt{histsize} is much larger than I originally thought. This could be increasing my span significantly and limiting my gains.

This lab was actually quite interesting. I could definitely spend at least twenty more hours exploring different tweaks to my implementation.
This lab was much more interesting. I decided to work on an improved mergesort as I’m rather fond of mergesort in general. My method was to initially parallelize the recursions, but then to also follow the suggested algorithm of section 27.3 of CLRS to further parallelize the merge. I outline first my design, then the difficulties I encountered, and then the results. The parallel merge if anything resulted in worse performance, but in doing this analysis I located some changes which could be made to improve it.

1 Design

The algorithm outlined for mergesort in CLRS was followed fairly closely. After parallelizing the recursive calls to mergesort, I proceeded to parallelize the merge operations themselves. In each merge of $A$ and $B$, assuming there are a number of elements greater than some threshold, the element at the midpoint of $A$ is selected. Then, using binary search (possible because the items to be merged are already sorted), the closest matching value in $B$ is found, partitioning each of $A$ and $B$. The first halves can then be merged together, as can the second halves.

2 Challenges

The first challenge was with memory management. The sorting functions expect to, on completion, be able to place the resulting sorted list in the place of the component lists which were sorted. However, when splitting the merges, the resulting list cannot be placed in the same space until both are done - doing so earlier with the result of one half would overwrite some portion to be sorted still being used by the other half’s thread. I got around this by copying a new space for results over for each fork of merge so they wouldn’t collide.

The second challenge was simply one of interface. I mistakenly thought each item to be sorted was a char, not realizing that each was really a struct containing a 64-bit value. All but the final test case worked assuming an int value stored in the char, but the last appeared to have negatives and I was unable to determine what was going on for several hours until a TA pointed the struct out. Thankfully, the code I’d written was already mostly correct aside from the data type, so correcting this left me with a mostly correct implementation.

3 Results

As seen in Figure 1 implementing a parallel merge function did not improve speed. In fact, the parallel merge function ran slower than the more simple parallelized mergesort. The span, according to cilkview, was decreased only 2% but the total work increased by 50%! This makes some sense given that most of the test cases were small and didn’t lend themselves to being further-divided, but is still surprising.
Figure 1: Times taken by the parallelized mergesort function, both with and without the merge portion parallelized. The more parallelized version runs substantially slower when doing row moves, but runs at nearly the same speed for pointers.
Looking at the difference between sorting with rows and pointers, there is not nearly as much performance loss with pointers. Given that the difference is one of memory use, this caused me to consider the difficulty I ran into with memory listed above. Because of its recursive nature, the parallel merge is actually copying all data twice as much as the serial version, which may explain the slowdown. I did not implement anything to deal with this, but believe I might have an answer. While the results really do need to be copied over because they change at every level, the arguments could be stored separately from where the result is placed, which would bring the parallel version down to the same amount of memory copying as is seen in the serial version, improving performance.