INF236 - Assignment 1

Kristian Sørdal

March 3, 2024

Contents

Problem 1

The implementation of sequential radix sort has been implemented in the following manner.

Algorithm 1: Sequential Radix Sort

```
**Input** : $n$ - The length of the array, $b$ - Key size (how many bits to interpret as one digit)
**Output:** $t$ - the time taken to sort the array
$a \leftarrow$ array of random unsigned 64-bit integers of size $n$
$tmp \leftarrow$ partially sorted array of size $n$, initialized with 0
$buckets \leftarrow 2^b$
**for** $shift \leftarrow 0$ **to** 64 **by** $b$ **do**
    $bucketSize \leftarrow$ array of size $buckets$, initialized with 0
    **for** $i \leftarrow 0$ **to** $n-1$ **do**
        $bucket \leftarrow (a[i] \gg shift)\&(buckets-1)$
        $bucketSize[bucket]{+}{+}$
    $sum \leftarrow 0;$
    $bucketSize[0] \leftarrow 0$
    **for** $i \leftarrow 1$ **to** $buckets - 1$ **do**
        $t \leftarrow sum + bs[i]$
        $bucketSize[i] \leftarrow t$
        $sum \leftarrow t$
    **for** $i \leftarrow 0$ **to** $n-1$ **do**
        $bucket \leftarrow (a[i] \gg shift)\&(buckets-1)$
        $tmp[bucketSize[bucket]] \leftarrow a[i]$
        $bucketSize[bucket]{+}{+}$
    $a \leftarrow tmp;$
```

Firstly, we allocate memory for a and tmp. a represents the input array, and will also contain the final sorted array, whereas tmp will store the partially sorted array during execution. a will be swapped with tmp every time the outermost for loop is done executing one iteration.

The outermost for loop iterates from 0 to 64, with a step size of b, where b represents how many bits should be interpreted as one digit. It iterates up to 64 bits, as this is the size of an `unsigned long long` type in `c`.

Each time this for loop iterates, it goes through 3 stages.

Stage 1 - Counting how many elements for each bucket In this stage, we iterate through the array containing our values. For each element x stored in $a[i]$, we want to figure out in which bucket this element should be placed into. Given s, representing the power we should raise x to, and k, representing how many buckets there are, we can obtain b, representing the bucket x belongs in. This can be done with the following formula

$$b = \left\lfloor \frac{x}{2^s} \right\rfloor \wedge k - 1$$

Which can be implemented in `c` as $bucket \leftarrow (a[i] \gg shift)\&(buckets - 1)$.

Stage 2 - Prefix sum In this stage, we need to perform a prefix sum operation on the *bucketSize* array. This is done in order to ensure we start placing the elements in the buckets at the correct index in the *tmp* array in the next stage.

Stage 3 - Creating the partially sorted array In this stage, we need to place our elements at the correct index in the *tmp* array. This is done by again, computing which bucket element $a[i]$ belongs to.

After we have found the bucket, we place this element in the *tmp* array, at index $bucketSize[bucket]$. Because we performed the prefix sum over the *bucketSize* array in the previous stage, this variable contains the index at which the elements belonging to this bucket should be placed in the *tmp* array. After we have placed element $a[i]$, we increment this index pointer, such that we avoid overwriting anything.

Problem 2

Through trial and error, the maximum elements that could be sorted in 10 seconds, was 60 million, with a value of $b = 4$. Varying b by powers of 2, we obtain the following execution times for sorting 60 million elements

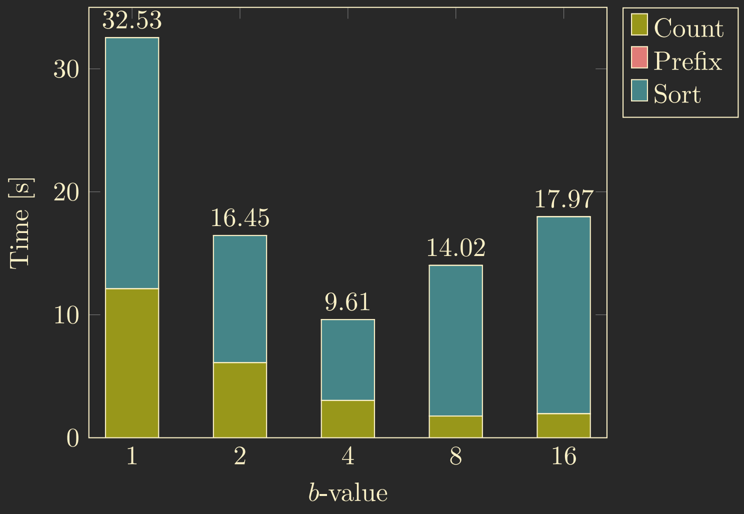

Figure 1: Sorting 60 million elements with varying b values. **NOTE:** Prefix value is so low its not visible.

As we can see, using a value b value of 4, yields the best running time. When we use $b = 4$, we have the option to place elements into $2^4 = 16$ different buckets. We also only have to perform the 3 stages described earlier $\frac{64}{4} = 16$ times.

Both the *Counting* and *Sorting* stage have a time complexity of $\mathcal{O}(n)$, and the *Prefix Sum* stage only has a time complexity of $\mathcal{O}(k)$, with k being the number of buckets. Because $k \ll n$ when n is sufficiently large, we don't see the impact of this stage in the plot, as it happens almost instantly.

But what explains the difference in execution time for the *Counting* and *Sorting* stage? To figure out why, we need to take a look at their implementation.

```
for (int i = 0; i < n; i++) {
    ull val = a[i];
    int t = (val >> shift) & (buckets - 1);
    bucketSize[t]++;
}
```

Figure 2: Counting stage

```
for (int i = 0; i < n; i++) {
    ull val = a[i];
    int t = (val >> shift) & (buckets - 1);
    permuted[bs[t]++] = val;
}
```

Figure 3: Sorting stage

As we can see, the implementation of the two stages are similar, but with some key differences. In the coutning stage, we fill the *bucketSize* to count how many elements are in each bucket. This array is of size 2^b, and is at most of size $2^{16} = 65536$. If we run the command `lscpu`, we cann see that the `L1d cachce` size is 32K, which means that the L1 cache can store up to 32KB, which is equivalent to 8192 integers, given that an integer is 4 bytes. This means that for all values of $b < 16$, the entire bucket array fits into this cache, allowing us to read and write to this array very fast. The size of the bucket array is also always a power of 2, which means that it will line up nicely within cache lines, which minimizes cache misses.

In the sorting stage however, we are writing to the *permuted* array, which is of size n. This means that for large values of n, this array does not fit nicely into L1 or even L2 cache (which can store up to 256KB), which results in slower memory access to this array. In addition, there is no ordering of which buckets are accessed when, which means that the array is written to in a random and sporadic pattern, which yields poor memory access patterns, further slowing us down.