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 52 million, with a value of $b = 4$. Varying b by powers of 2, we obtain the following execution times for sorting 52 million elements

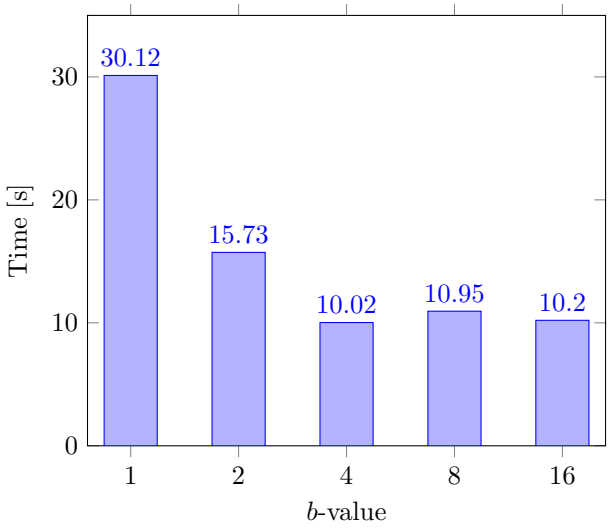

Figure 1: Sorting 52 million elements with varying b values