INF236 - Assignment 1

Kristian Sørdal

March 3, 2024

Contents

1 File Overview

main.c	Main Executable
radix_seq.c	Sequential Radix Sort
radix_par.c	Parallel Radix Sort
util.c	Various utilization functions and definitions
mt19937-64.c	Random Number Generator

Please note that most files have header files that contain function declarations.

IMPORTANT: Instead of having to compile two different files for the sequential and parallel version, when you compile the file `main.c`, just add the compile flag `-DPAR` if you want to compile the parallel version, or the compile flag `-DSEQ` if you want to compile the sequential version.

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

b-value	Count [s]	Sort [s]	Total [s]
1	12.115	20.418	32.533
2	6.105	10.341	16.446
4	3.032	6.576	9.608
8	1.765	12.249	14.015
16	1.951	16.018	17.969

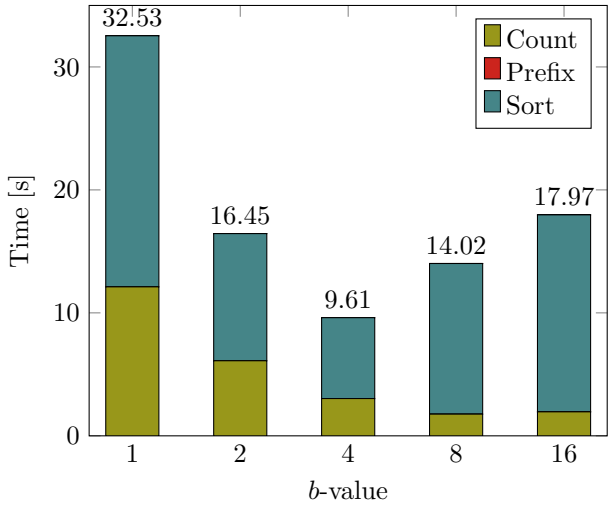


Figure 1: Sorting 60 million elements with varying b values. **NOTE:** Prefix value is so low its not visible.

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.

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

```
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

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.

Computing an expression for execution time In order to compute an expression for the execution time, we can start by looking at the runtime. The runtime of radix sort, expressed in big O notation is $\mathcal{O}(n \cdot k)$, where n is the size of the array, and k is the key size, or how many bits to interpret as one digit. Taking a closer look at the runtime, and including constants, we can see that the three stages have the following runtimes:

$$\text{Counting} : 3n$$
$$\text{Prefix Sum} : 2 \cdot 2^k$$
$$\text{Sorting} : 3n$$

So the running time of the algorithm, including constants is $\mathcal{O}\left(k \cdot \left(6n + 2 \cdot 2^k\right)\right)$. This does not tell us much about how long the algorithm will take to complete. To figure this out, we need to benchmark the different stages. We are specifically interested in how long one iteration of the *Counting* stage, and the *Sorting* stage will take. We will disregard the time for the prefix sum, as it runs close to instantaneously for all possible values for b.

In order to benchmark these loops, our best bet is to time each execution of the two loops individually, and put this time into two arrays of size $\frac{64}{b}$, then accumulate this time and divide it by $b \cdot n$. Mathematically, this can be expressed as

$$t_{avg} = \frac{t_{tot}}{n \cdot b}$$

Where t_{tot} is the total time for b executions of the loop. By runnning the sequential program with $n = 1 \times 10^7$, and $b = 8$, we obtained the time of execution for one iteration of the *Counting* loop and *Sorting* loop.

$$t_{count} = 3.6355 \times 10^{-9} s, \quad t_{sort} = 2.3273 \times 10^{-8} s$$

Then, because these loops will be executed $\frac{64}{b}$ times, we can express the execution time of the algorithm as

$$
\begin{aligned}
T(n,b) &= \frac{64}{b} \cdot (n \cdot t_{count} + n \cdot t_{sort}) \\
&= \frac{64n}{b} \cdot (t_{count} + t_{sort}) \\
&= \frac{64n}{b} \cdot \left(3.6355 \times 10^{-9} + 2.3273 \times 10^{-8}\right) \\
&= \frac{64n}{b} \cdot 2.69085 \times 10^{-8} s
\end{aligned}
$$

Now that we have this expression, we can compare it to the actual execution time of the algorithm, and see how well it holds up.

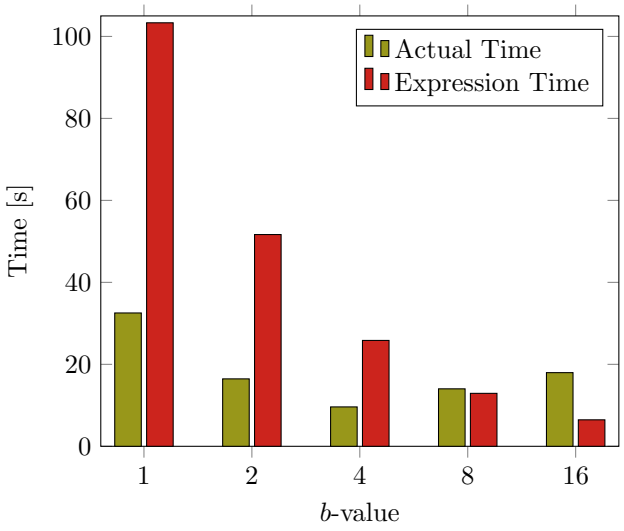


Figure 4: Comparison between execution time of sequential algorithm according to expression and actual execution time.

Unsurprisingly, the expression isn't accurate at all, as the values obtained for t_{count} and t_{sort} is different every time the algorithm is ran, and so it is impossible to determine an accurate value for these variables which is general enough to hold for all combinations of n and b. It should also be noted that the value for these variables were obtained from a program compiled with `-O3`, which may introduce unforseen optimizations that make the expression dubious at best.

Problem 3

The implementation of the parallel algorithm is similar to the sequential version. This section will highlight the differences, and what makes it faster. Primarily, the *Counting* and *Sorting* stage benefit the most from parallelization, whereas the *Prefix Sum* stage has been slightly modified.

Before we start sorting the array, we first compute the parts of the list which each thread shall perform work on. This is done using the following algorithm:

Algorithm 2: Computing Ranges for each thread

Input : *begins* - array to store starting indices, *ends* - array to store ending indices, n - the size of the array, p - the number of threads
Output: *begins* and *ends* filled with starting and ending indices, respectively
$begins[0] \leftarrow 0$
for $i \leftarrow 0$ **to** $p - 1$ **do**
  $ends[i] \leftarrow (i + 1) \cdot \frac{n}{p}$
  $begins[i + 1] \leftarrow ends[i]$
$ends[p - 1] = n$

Next, we initialize a histogram of size $p \cdot 2^b$, which will count the number of elements in each bucket.

After this is done, we start iterating through the main loop, going in increments of b, as in the sequential version. We then enter a parallel region, in which each thread works on its designated portion of the array, such that each thread only has to to $\frac{n}{p}$ iterations over the loop, as opposed to n iterations in the sequential version. During this iteration, each thread will fill the histogram. Specifically it will will the row in the histogram according to its thread ID.

Moving on, we perform a column-wise prefix sum on the histogram array. This is done sequentially, as the histogram is relatively small when compared to n, and so there is nothing to be gained from doing it in parallel. The reason we have to to the prefix sum column-wise and not row-wise, is that the values in the histogram needs to be a pointed representing where thread x should start placing elements belonging to bucket y, and because each thread has elements in bucket y, we need to iterate column-wise.

After the prefix sum, we move on to another parallel section, where we will fill up the permuted array with values from the original array. This is done with the same logic as in the sequential version, except that each thread only has to to $\frac{n}{p}$ iterations, and that each thread again use the pointers from the row in the histogram corresponding to their thread ID. When this stage is done, we swap the pointers of the original array and the permuted array, and continue sorting.

There has been made one other minor optimization to this algorithm when compared to the sequential version, and that is the utilization of aligned memory allocation. When the command `getconf LEVEL1_DCACHE_LINESIZE` is ran on Brake, we obtain the length in bytes of each cache line in the L1 cache. This happens to be 64, which is why we have the `CACHE_LINE_SIZE 64` define in the `radix_par.c` file. We can then use the following function to align memory so that it fits better in these cache lines, using the following function from `radix_par.c`:

```
void *aligned_alloc_generic(size_t size, size_t num_elements, size_t element_size) {
    void *block = NULL;
    if (posix_memalign(&block, size, num_elements * element_size) != 0) {
        fprintf(stderr, "Failed to allocate memory\n");
        exit(EXIT_FAILURE);
    }
    return block;
}
```

Computing an expression for execution time In order to compute an expression for the execution time for the parallel algorithm, we have to introduce another argument, that being p - the number of threads. We can use the same expression as we did for the sequential expression, but we have to exchange n with $\frac{n}{p}$. This yields the following expression

$$T(n, b, p) = \frac{64\frac{n}{p}}{b} \cdot 2.69085 \times 10^{-8} s$$

Comparing this expression to the actual runtime, we get the following table and graph

# Processors	Actutal time [s]	Expression time [s]
1	9.461611	25.83
10	1.813418	2.583
20	1.983802	1.2916
30	1.952954	0.8610
40	1.857981	0.6458
50	2.301503	0.5166
60	1.663963	0.4305
70	2.477585	0.36903
80	2.611293	0.3229

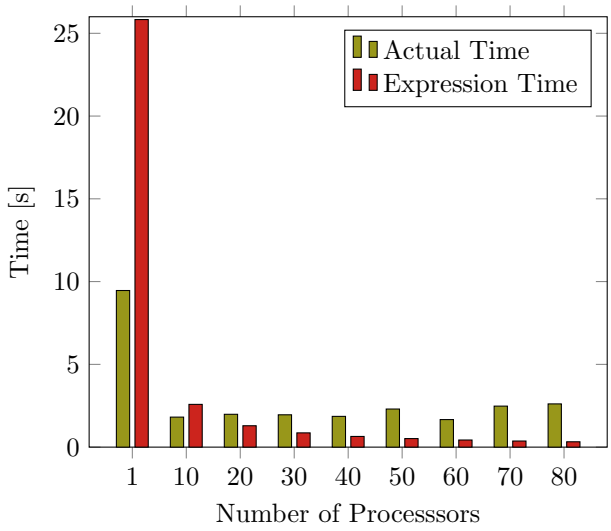
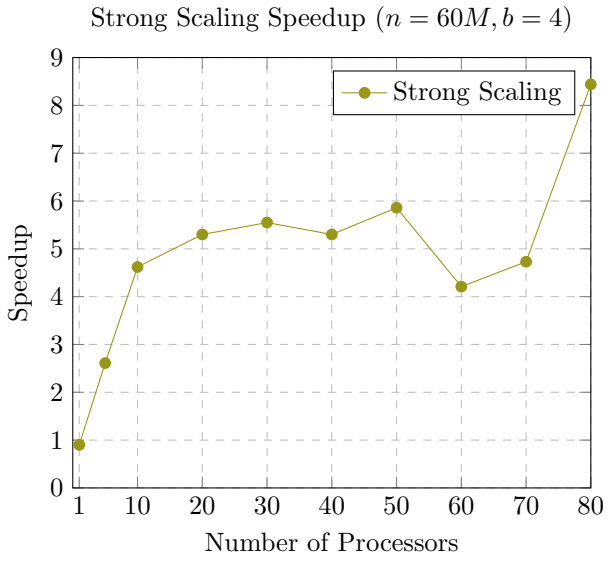


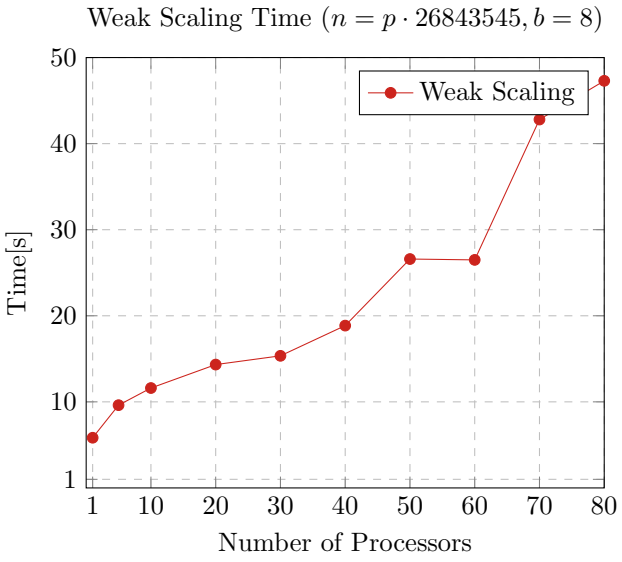
Figure 5: Comparison between execution time of parallel algorithm according to expression and actual execution time.

As we can see, the expression does not fit very well with the actual running time at all. The expression we obtained is very close to optimal speedup, and clearly our parallel algorithm does not achieve that. This is due to a myriad of reasons, such as poor memory access patterns, overhead of sequential work, overhead of intializing parallel environments, cache misses, and even not getting enough resources on the machine the code was ran on.

As a final note before strong and weak scaling, it should be noted that I attempted to implement some kind of buffer system for the parallel algorith, as the initialization of the permutation array was the major bottleneck. I wanted to try to give each thread a buffer to fill elements in sorted order, and when it reached a certain size, flush this buffer into the sequential array. I was however unsuccessful in implementing this.

Problem 4





The number 26843545 as chosen for the weak scaling, as this was the maximum workload per thread when using 80 threads. This resulted in approximately $n = 2147483600$ when using 80 threads. Please note that resources on Brake were sparse during some of these tests, so data may not be 100% accurate.