micksort: Parallelizing Counting Sort on MPI

Micky Faas

December 31, 2018

1 Introduction

Counting sort is a trivial sorting algorithm that counts the occurrence of unique elements in the input sequence A in order to produce a sorted sequence B. The result is not directly a sorted sequence, but a function f that maps input elements to their prevalence in the original sequence A. Both f from A and B from f can be constructed in linear time, which gives counting sort an overall complexity of $O(n)$. For some large sequences, there are cases were f can be stored in a much smaller memory space than either A or the sorted sequence B. In these cases the transformation of A to f creates an implicit form compression, which is one of the advantages of counting sort that we will later exploit to limit communication overhead.

Despite these advantages, counting sort is not used very often because it imposes many constraints on the form of the input sequence A. The first and most obvious limitation is that the space of elements in A can somehow be mapped to $\mathbb{N}$, i.e. it limits the use cases to elements that can be represented by integers. Because of this slight generalization we will talk about counting *bins* rather than numbers. This assumes that some function g is present that maps the type of the elements in A to its corresponding, unique bin index. In the trivial case where A contains just integers, this function will probably look like $g : \mathbb{N} \mapsto \mathbb{N}$. The range that these bin indices are on, directly determines the storage size of the function f, which is often implemented as a linear array. Implementations with hash-maps are also possible, but lead to worse performance in case of collisions. This means that some information about the input sequence is required and thus counting sort is impractical as a general purpose algorithm.

Another important aspect is that the average and maximum counts for any given element in A are key to the overall performance of counting sort. The maximum count determines the size of the bins. If the distribution of A is very non-uniform, this may lead to very large bins of which only few are filled. The average count is important to the efficiency of the storage. For low average counts this means that bins are mostly under-filled and that the range of indices is spread out considerably. This means that the ratio of $|A|$ to the amount of space it takes to store f is not favourable. On the other hand, if the average count is high, it means that elements are distributed rather uniform and that the range is quite small. In these cases, we will show that using counting sort may lead to a significant improvement of both complexity and storage complexity compared to other algorithms.

1.1 Use Case

In order to demonstrate a parallel counting sort, the following use case will be defined. The input sequence will consist of pseudo-random numbers generated by `LIBC`'s `rand()`. We will generate the numbers directly on the individual nodes with no intermediate storage. The total length of the entire sequence is varied, as is the number of physical nodes. Sequences of 200K, 1,6M, 80M, 16B, 80B and 160B will be tested on 1, 2, 4, 8 and 16 compute nodes. Here one compute node means one single-threaded machine from the Leiden DAS-4 cluster. Instead of accessing all elements of the sorted sequence B, we will only access every 10,000th element.

2 Parallel Counting Sort

The basis for counting sort is the native, sequential notation shown in Algorithm 1. We could easily subdivide A into multiple sequences and process these in parallel. However, multiple nodes would then have to access some $f(i)$ simultaneously, while the range of i cannot be predicted beforehand. To alleviate this we could pre-sort the input sequence into ranges, such that the individual processes each get a subset of $f()$ and the corresponding range of element in A. This would be a viable option for data that is already stored to disk. For the random numbers, it proved to be faster in practice to duplicate the complete $f()$ to each process and not pre-sort A at all.

Algorithm 1 Naive counting sort

```
function COUNTSORT($A$)
    $f(i) \leftarrow 0 \ \forall \ i \in \{\min(A), \dots, \max(A)\}$
    for all $a \in A$ do                                ▷ Generate $f()$
        $f(a) \leftarrow f(a) + 1$
    end for
    $i \leftarrow \min(A)$
    while $i < \max(A)$ do                              ▷ Generate sorted sequence $B$
        $j \leftarrow 0$
        while $j < f(i)$ do
            Append $i$ to $B$
        end while
        $i \leftarrow i + 1$
    end whilereturn $B$
end function
```

Another concern for the naive approach is that in typical implementation $f()$ is stored as a linear array. If the number of bins is large enough, this array will not easily fit in cache. Since the access pattern is mostly random, memory performance will be low as no adequate caching can occur.

Despite the obvious shortcomings, we have still utilized the naive approach in the parallel formulation of the algorithm. The simple reason for this is that empirical testing showed that more complex approaches always degraded the performance for our specific use-case and the space required for stored intermediate pre-sorted sequences out weights the benefit to be gained from better caching.

Algorithm 2 Parallel naive counting sort

```
function COUNTSORT($A$, $np$, $p$)    ▷ Executed in parallel on process $p$ of $np$
    $N \leftarrow \max(A) - \min(A)$    ▷ Number of bins
    $N_{pp} \leftarrow \frac{|A|}{np}$    ▷ Number of elements per process
    $f(i) \leftarrow 0 \; \forall \; i \in \{0, \ldots, N\}$
    for all $a \in \{A(pN_{pp}), \ldots, A((p+1)N_{pp}) - 1\}$ do    ▷ Generate $f()$
        $f(a) \leftarrow f(a) + 1$
    end for
    $N_{bp} \leftarrow \frac{N}{np}$    ▷ Bins per process
    $i \leftarrow 0$
    while $i < np$ do    ▷ Distribute ranges of $f()$ over all processes
        Send $\{f(iN_{bp}, \ldots, f((i+1)N_{bp} - 1\}$ to process $i$
        Receive $f_i()$ from process $i$
        $i \leftarrow i + 1$
    end while
    Merge $f_0(), \ldots, f_{np-1}()$ into $f'()$ with $N_{bp}$ bins
end function
```

The Algorithm 2 consists roughly of three steps. The first step divides A into np parts of which each process sorts its own part. For this, an $f()$ of length N is required on each process. In the second step communication takes place. Important is that each process only receives a small part of every other process' $f()$ function. More specifically, each process p sends its lowest N_{bp} bins to process 0 its second-to-lowest bins to process 1 and so on. In the end each process posesses np different $f()$ functions of length N_{bp} with the same bin indices. These functions are merged in the last step, where all bins are summed to produce one count function with the range $[pN_{bp}; (p+1)N_{np})$. This means that each node can only produce a limited range of the sorted sequence.

Although the communication choices made in this algorithm may seem impractical, it solves an important scaling problem. If each process would communicate the entire count function $f()$ to every other process, the size of the communication would increase exponentially as more process are added. For this algorithm, each process sends and receives exactly N times the bin size, regardless of the number of processes.

3 Experiments

In addition to the parameters defined by the use case, two more variables were tested during the experiments. First of all it is important to note that the calls to `rand()` cannot be separated from the sorting process. The reason for this is that no intermediate storage is used to improve performance. It appeared that the GLIBC `rand()` function caused so much overhead, that it was the bottleneck for the naive sequential counting sort algorithm. The manual pages for `rand()` state that a trivial congruential PRNG is utilized, but upon inspection of the GLIBC sources, `rand()` actually calls the BSD-style `random()` function which uses a vastly more complicated PRNG. We therefore tested the GLIBC `rand()`

Processes	16B	80M	1,6M	200K
1	810.919s	4.6143s	0.594061s	0.52114s
2	407.088s	4.12661s	2.09625s	2.06978s
4	204.858s	3.04667s	2.04611s	2.01834s
8	104.902s	2.47457s	2.04964s	1.91673s
16	54.0518s	3.42794s	3.36872s	3.17901s
Input size	59.6 GiB	305 MiB	6.1 MiB	781.3 KiB

Table 1: Timings for byte-sized bins and GLIBC `rand()`

Processes	160B	80B	16B	80M
1	2063.13s	1034.98s	206.816s	1.57739s
2	1033.4s	519.105s	106.391s	3.26186s
4	520.176s	260.197s	53.7344s	3.09173s
8	261.439s	132.264s	28.8188s	3.06876s
16	132.614s	67.8465s	16.1833s	3.22445s
Input size	596 GiB	298 GiB	59.3 GiB	305 MiB

Table 2: Timings for byte-sized bins and fast random

against the old PRNG defined by POSIX. This can be seen by comparing Table 1 and 2.

The second question was whether the size of the array used to store $f()$ had any effect on performance. One would expect that both cache misses and communication overhead decrease when this array is smaller. We therefore tested with byte-sized bins (Tables 1 and 2) and 4-bits bins (Table 3). Due to the uniform characteristic of the PRNG, 4-bit bins suffice up to sequences of 80M 32-bit integers. Given that `rand()` produces numbers in the range $[0; 2^{31} - 1]$, the arrays were sized 2 GiB (byte bins) and 1 GiB (nibble bins). Also notice how these sizes are vastly smaller than the sizes of the larger sequences.

3.1 Conclusion

From the results can be seen that it makes very little sense to use parallel counting sort for the sequences under 80M. The communication overhead is bigger than the gain from parallelization while the excess storage required by the algorithm makes little sense. Even for 80M the gain is far from linear. For very large sequences, however, scaling becomes very linear and the counting function takes (much) less space that the original input sequence. For sequences over 16B it was infeasible to use `rand()` and therefore we resorted to an inlined

Processes	80M	1,6M	200K
1	4.28085s	0.341267s	0.266851s
2	3.52983s	1.52962s	1.49207s
4	2.40327s	1.4418s	1.42061s
8	1.93671s	1.53077s	1.49064s
16	1.91732s	1.61514s	1.61439s

Table 3: Timings for 4-bit bins and GLIBC `rand()`

POSIX congruential PRNG. This shows that the counting sort mechanism can be so fast that the retrieval or computation of the input data becomes the bottleneck.

Of course the described algorithm works well only for specific instances such as these. However these results show that counting sort may be worth using if the problem happens to fit within these narrow constraints.