Parallel Count-Min Sketch on Multicores

Fatih Taşyaran, Kerem Yıldırır, Mustafa Kemal Taş and Kamer Kaya
Faculty of Engineering and Natural Sciences,
Computer Science and Engineering
Sabancı University, Turkey
E-mail: {fatihtasyaran, keremyildirir, mkemaltas, kaya}@sabanciuniv.edu

***Abstract*—One of the biggest challenges today is analyzing enormous chunks of data gathered from different sources. Traditional approaches for data analysis fail to scale as their memory and time requirement are too high to be practical. Sketches are probabilistic data structures that can provide approximate results within mathematically proven error bounds while having orders of magnitude less memory and time complexities than traditional approaches. Hence, they can be employed for big data analysis by architectures even with limited memory such as a `Raspberry Pi`. Today, such architectures usually offer multiple cores. With efficient parallel sketching, they can manage high volumes of data streams and be utilized for edge computing. For instance, a single Raspberry Pi deployed at the edge of a network can perform frequency analysis on streaming data obtained by many IoT sensors.**

In this work, we provide a parallel implementation for Count-Min Sketch which has been frequently used for various applications. We evaluate its performance both on a high-end server and a Raspberry Pi. We used tabulation hashing and to obtain a parallel and cache friendly implementation, we combine the tables for different hash functions in an alternating way. The results are promising as they show that the proposed implementation can be applied to improve the amount of real-time streaming data managed by single board computers and higher-end servers with multiple cores.

I. Introduction

Today, many applications such as financial markets [7] and network traffic analysis [15], [16] has a streaming nature. For these applications, the data can rapidly scale up to massive amounts, making them impossible to store and allowing only a single pass over the stream. Furthermore, since the data arrive at real-time, it may be impractical to process it and answer the queries, such as membership [4] and/or frequency [8], with 100% accuracy. Although exact computation can be possible for slower streams by processing the data on cutting-edge servers equipped with a large memory and powerful processor(s), enabling less power hungry devices such as single-board computers, e.g., Raspberry Pi, Odroid etc., with smarter algorithms and data structures yields much energy efficient and cheaper solutions for the same task. A comprehensive survey of data stream applications can be found in [19].

Sketching is a probabilistic data summarization technique; there exist various sketches in the literature tailored for different applications. These data structures can work on streaming data and help us to process a query with small, even sub-linear, amount of memory for many problems [1], [8], [14], [17]. Furthermore, sketches are very useful for distributed data streams since each stream can independently be sketched and then these sketches can be combined to obtain the final sketch. To remove the bias on the data distribution, sketches frequently employ hashing. This maps the actual data to datasets on which various statistical analysis can be performed for problems such as heavy-hitters, count distinct, rare counts etc. Due to the implicit compression being performed during this process, there is almost always a trade-off between the accuracy of the final result and the size of the sketch. A complete analysis and comparison of various sketches can be found in [11].

As stated above, sketches are tailored for different applications; one of the most popular and well-known sketches today is *Bloom Filter* [2] which is designed to answer membership queries. Despite allowing false-positives, i.e. falsely confirming the existence of a non-existent item, Bloom Filters have proven to be useful for many applications employing membership queries [3]. However, they fail to perform any kind of counting the frequencies of the items. Thus, a more sophisticated data structure is required.

Count-Min Sketch (CMS) is probabilistic sketch that helps to estimate the frequencies, i.e., the number of occurrences, of distinct items in a stream [11]. The frequency information is crucial to find finding heavy-hitters or rare items or detecting anomalies within a large computer network [9], [11]. Unlike Bloom Filters which use bits, CMSs store a small counter table to keep track of the frequency of each distinct element. Intuitively, the reduction to sub-linear space decreases the accuracy in the sense that the frequencies of some elements can be overestimated due to the collisions of hash functions. An important property of a CMS is that the error is always one sided; that is, the sketch never underestimates the frequencies.

In this work, we focus on multicore implementation of Count-Min sketches and experiment both on a high-end server and a Raspberry Pi. As hashing function, we used *tabulation hashing* which is simple and provide unexpectedly strong statistical guarantees [20], [23]. Our first contribution is designing a cache-friendly implementation to compute multiple tabulation hashes at a time which is a crucial operation for CMS. The proposed approach can be useful not only for CMS but also other applications that needs to hash the same element multiple times which is a common operation in many sketches. We then restructure the CMS construction in a way to process the data stream in parallel while avoiding possible

race-conditions on a single CMS table as much as possible. An efficient synchronization mechanism is necessary since race-conditions can both decrease the performance and also increase the amount of error estimated by statistical analysis by taking the table size into account.

The rest of the article is organized as follows: Section II presents the background and describes the notation that will be used later. In Sections III and IV, the cache-friendly tabular hashing mechanism and corresponding parallel CMS construction implementation will be proposed. Section V presents the experimental results and Section VII concludes the paper.

II. Notation and Background

Let $\mathcal{U} = \{1, \cdots, n\}$ be the universal set where the elements in the stream are coming from. Let N be size of the stream $\mathtt{s}[.]$ where $\mathtt{s}[i]$ denotes the ith element in the stream. We will use f_x to denote the frequency of an item. Hence,

$$f_x = |\{x = \mathtt{s}[i] : 1 \leq i \leq N\}|.$$

Given two parameters ϵ and δ, a Count-Min Sketch is constructed as a two-dimensional counter array with $d = \lceil \ln(1/\delta) \rceil$ rows and $w = \lceil e/\epsilon \rceil$ columns. There are two fundamental operations for a CMS; the first one is *insert*(x) which is used to construct/update the internal sketch counters to insert an item $x \in \mathcal{U}$. Initially, all the counters inside the sketch are set to 0. To insert an item $x \in \mathcal{U}$, for $1 \leq i \leq d$, the counters $\mathtt{cms}[i][h_i(x)]$ are incremented. Figure 1 illustrates a single insertion and Algorithm 1 gives the pre the process of processing a stream $\mathtt{s}[.]$ of size N.

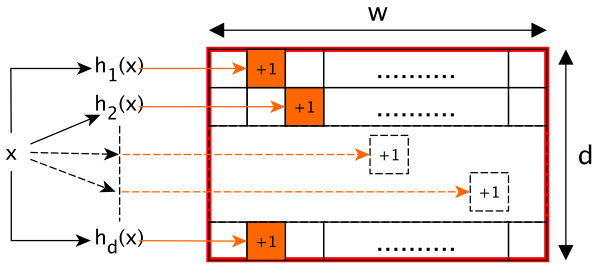


Fig. 1: Illustration of the insert operation for Count-Min Sketch

The second operation for CMS is *query*(x) which estimates the frequency of $x \in \mathcal{U}$ as

$$f'_x = min_{1 \leq i \leq d}\{\mathtt{cms}[i][h_i(x)]\}.$$

With $d \times w$ memory, the sketch satisfies that $f_x \leq f'_x$ and $f'_x \leq f_x + \epsilon N$ with probability at least $1 - \delta$. Hence, the error is always one-sided, i.e., the estimation is always larger than the actual frequency. Furthermore, the error is additive, and for ϵ and δ small enough, the amount of the error is also small w.h.p. especially if f_x is large, i.e., x is a frequent item.

A. Tabulation Hashing

CMS uses pairwise independent hash functions to provide the desired properties stated above. There exist a separate hash function for each row of the CMS with a range equal

Algorithm 1 CMS-Construction

Input: ϵ: error factor
δ: error probability
$\mathtt{s}[.]$: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \leq i \leq d$, $h_i: \mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$

Output: $\mathtt{cms}[.][.]$: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$

```
1: for i from 1 to d do
2:     for j from 1 to w do
3:         cms[i][j] ← 0
4: for i from 1 to N do
5:     x ← s[i]
6:     for j from 1 to d do
7:         col ← h_j(x)
8:         cms[j][col] ← cms[j][col] +1
```

to the range of columns. In this work, we use Tabulation Hashing [24] which has been recently analyzed by Patrascu and Thorup et al. [20], [23] and shown to provide unexpectedly strong guarantees despite of its simplicity. Furthermore, it has been shown that even a slightly more sophisticated version of tabulation hashing is almost as fast as the classic multiply-mod-prime scheme, i.e., $(ax + b) \bmod p$.

Assuming each element in U is represented in 32 bits and the desired output is also 32 bits, the tabulation hashing works as follows: first a 4×256 table is generated and filled with random 32-bit values. Given a 32-bit input x, each character, i.e., 8-bit value, of x is used as an index for the corresponding row. Hence, four 32-bit values, one from each row, are extracted from the table. The XOR of these 32-bit values are returned as the hash value. The implementation we use in this work is given in Figure 2.

```
uint32_t hash(uint32_t x, uint32_t table[4][256]) {
  uint32_t h = 0;
  for (uint32_t i = 0; i < 4; i++) {
    uint32_t c = x & 0x000000ff;
    h ^= table[i][c];
    x = x >> 8;
  }
  return h;
}
```

Fig. 2: Naive tabulation hashing in C++

III. Multiple Tabulation Hashes at Once

Although the auxiliary data used in tabulation hashing are usually small and can fit into a cache, the spatial locality of the accessed elements in the tables are not cache friendly since each access is performed to a different table row. A naive, cache-friendly rearrangement of the entries in the table is also not possible for applications performing a single hash per item; the indices for the table rows are obtained from adjacent chunks in the binary representation of the hashed item whose decimal values are usually not correlated hence there is no

relation whatsoever among them to help us to fix the access pattern.

In addition to CMS, for many frequency sketches, e.g., Count Sketch [8] and Count-Mean-Min Sketch [18], the same item is hashed more than once. For tabulation hashing, this yields an interesting optimization; there exist multiple hash functions and hence more than one hash table. Although, a single table is still accessed in an irregular fashion, the accessed coordinates in all the tables are the same for different tables. This can be observed on the left side of Figure 3. Hence, the columns of the tables can be combined in an alternating fashion as shown in the right side of the figure. In this approach, when only a single thread is responsible from inserting the appearance of a single item to CMS, the cache can be utilized in a better way. This is called *merged tabulation*.

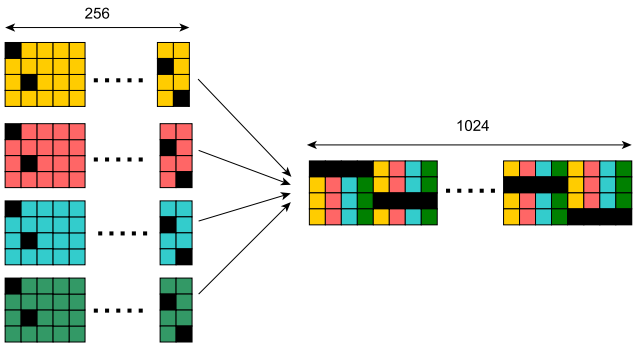


Fig. 3: Memory access patterns for naive and merged tabulation for four hashes. The hash tables are colored with different colors. The accessed locations for both approaches are shown in black.

With merged tabulation, the consecutive elements in the merged table belong to different hash values. Since these values require multiple XOR operations, we need to store the partial results in the memory. A simple C++ implementation of merged tabulation which uses a single 4×1024 table to compute four hashes at once is given in Figure 4.

IV. Parallel Count-Min Sketch Construction

Since multiple CMS sketches can be combined, on a multicore hardware, each thread can process a different part of the data (with the same hash functions) to construct a partial CMS. These partial sketches can then be aggregated by adding the counter values in the same locations to obtain the final value on the combined CMS sketch. Although this approach requires no synchronization until all partial sketches are constructed, compared to constructing the final CMS table directly, it consumes more memory. Furthermore, the amount of the memory is increasing with increasing number of threads.

Constructing a single CMS sketch in parallel is not a straightforward task. The pseudocode of the naive parallel CMS construction is given in Algorithm 2. When a thread processes a single data item, it accesses an arbitrary column

```
void hash(uint32_t x, uint32_t hashes[8],
          uint32_t table[4][1024]) {
  uint32_t c = (x & 0x000000ff) << 2;
  x = x >> 8;

  hashes[0] = table[0][c];
  hashes[1] = table[0][c + 1];
  hashes[2] = table[0][c + 2];
  hashes[3] = table[0][c + 3];

  for (uint32_t i = 1; i < 4; i++) {
    c = (x & 0x000000ff) << 2;
    x = x >> 8;

    hashes[0] ^= table[i][c];
    hashes[1] ^= table[i][c + 1];
    hashes[2] ^= table[i][c + 2];
    hashes[3] ^= table[i][c + 3];
  }
}
```

Fig. 4: Merged tabulation hashing implementation in C++

of each CMS row. Hence, race conditions may reduce the estimation accuracy. In addition, these memory accesses are a probable cause of false sharing which prevents the implementation to utilize the hardware to its full potential. To avoid the pitfalls stated above, one could assign each item to multiple threads which evaluate different hash functions hence touch different rows of the CMS table. However, this approach cannot use the merged tabulation since, each thread performs only a single hash of an item.

Algorithm 2 Naive-Parallel-CMS

Input: ϵ: error factor
δ: error probability
$\mathtt{s}[.]$: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \leq i \leq d$, $h_i \colon \mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$
τ: no threads
Output: $\mathtt{cms}[.][.]$: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$

1: Reset all the $\mathtt{cms}[.][.]$ counters to 0 (as in Algorithm 1).
2:
3: **for** i from 1 to N **in parallel do**
4: $x \leftarrow s[i]$
5: $\mathtt{hashes}[.] \leftarrow$ MergedHash(x)
6: **for** j from 1 to d **do**
7: $col \leftarrow \mathtt{hashes}[j]$
8: $\mathtt{cms[j][}col\mathtt{]} \leftarrow \mathtt{cms[j][}col\mathtt{]} + 1$

In this work, we propose a *buffered parallel* execution to alleviate the above mentioned issues; we (1) divide the data into batches and (2) process a single batch in parallel in two stages; a) hashing and b) CMS counter updates. In the proposed approach, after each batch, the threads synchronize and process the next one. For batches with b elements, the first stage requires a buffer of size $b \times d$ to store the hash values, i.e., column ids, which then will be used in the second stage to update CMS counters. Such a buffer allows us to use merged tabulation effectively during the first stage. For the second

stage, we used at most d threads. In our implementation, the counters in a row are updated by the same thread hence, there will be no race conditions and probably much less false sharing. We also experiment with extra padding between the table rows which we found to be useful for CMS tables with a small w value. Algorithm 3 gives the pseudocode for our CMS construction approach.

Algorithm 3 BUFFERED-PARALLEL-CMS

Input: ϵ: error factor
δ: error probability
$\mathtt{s}[.]$: a stream with N elements from n distinct elements
$h_i(.)$: pairwise independent hash functions where for $1 \leq i \leq d$, h_i: $\mathcal{U} \rightarrow \{1, \cdots, w\}$ and $w = \lceil e/\epsilon \rceil$
b: batch size (assumption: divides N)
τ: no threads (assumption: divides d)
Output: $\mathtt{cms}[.][.]$: a $d \times w$ counter sketch where $d = \lceil 1/\delta \rceil$

```
1: Reset all the cms[.][.] counters to 0 (as in Algorithm 1).
2:
3: for i from 1 to N/b do
4:     j_end ← i × b                                          ▷ Stage 1
5:     j_start ← j_end − b + 1
6:     for j from j_start to j_end in parallel do
7:         x ← s[j]
8:         ℓ_end ← j × d
9:         ℓ_start ← ℓ_end − d + 1
10:        buf[ℓ_start, ··· , ℓ_end] ← MERGEDHASH(x)
11:    for t_id from 1 to τ in parallel do                    ▷ Stage 2
12:        for j from 1 to b do
13:            nrows ← d/τ
14:            r_end ← t_id × nrows
15:            r_start ← r_end − nrows + 1
16:            for r from r_start to r_end do
17:                col ← buf[((j − 1) × d) + r]
18:                cms[r][col] ← cms[r][col] + 1
```

A straightforward parallelization with multiple sketches uses $(d \times w \times \tau)$ counters where each counter can have a value as large as N. Hence, the memory consumption of the naive approach is $(d \times w \times \tau \times \log(N))$ bits. On the other hand, a single sketch with buffering consumes $(d \times (w \times \log(N) + b \times \log(w)))$ bits since there are $(d \times b)$ entries in the buffer and each entry is a column id on CMS.

V. EXPERIMENTAL RESULTS

To understand the performance of the proposed approach, we perform the experiments on two architectures: **Arch-1** is a server running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). **Arch-2** is a quad-core 64-bit ARM Cortex A53 clocked at 1.2 GHz equipped with 1 GB LPDDR2-900 SDRAM. It is running on Raspbian GNU/Linux 9. For the multicore implementations, we use C++11 and OpenMP. We use `gcc 5.3.0` with the `-O3` optimization flag enabled on **Arch-1**. On **Arch-2** the compiler is `gcc 6.3.0` and the `-O3` optimization flag is enabled.

To generate the datasets used in the experiments, we used a *Zipfian* distribution. Although this study focuses on the performance which is not directly related to the distribution of the values in the stream $\mathtt{s}[.]$, many data in real world such as number of paper citations, file transfer sizes, word frequencies etc. fit to a Zipfian distribution. Furthermore, although the statistical analysis for CMS do not depend on the data, Zipfian is a common choice for the studies in the literature to benchmark the estimation accuracy of data sketches. We used various shape parameters $\alpha \in \{0.9, 1.1, 1.5, 2.0\}$ to generate the streams and to analyze the impact of frequency distribution on the performance.

We employ two sets of (ϵ, δ) values to generate CMS tables; in the first one, we used $(\epsilon = 0.01, \delta = 0.003)$ to generate a **small** sketch with $d = \lceil \log_2(1/\delta) \rceil = 8$ and $w = 211$ where w is selected as the first prime after $2/\epsilon$. In addition, we generate a relatively **large** sketch with $(\epsilon = 0.01, \delta = 0.003)$ with $d = 8$ rows and $w = 2003$ columns.

For the experiments on **Arch-1**, we choose $N = 2^{30}$ elements from a universal set $\mathcal{U}$ of cardinality $n = 2^{25}$. For **Arch-2**, i.e., Raspberry Pi, we use $N = 2^{25}$ and $n = 2^{20}$. We used $b = 1024$ as the batch size. Each data point in the tables and charts given below is obtained by averaging ten consecutive runs.

With these parameters, assuming 8 threads and $N = 2^{30}$, the naive approach with multiple sketches require $(8 \times 2003 \times 8 \times 30)$ = 3.85 Mbits whereas the proposed approach requires $(8 \times (2003 \times 30 + 1024 \times 11))$ = 570 Kbits. For $N = 2^{25}$, these numbers are 3.2 Mbits and 61 Kbits, respectively.

A. Parallel CMS construction with partial sketches

In the first set of experiments, we measure the performance improvement due to merged tabulation in the partial sketches setting. That is, each thread is responsible from a different part of the stream and construct a partial sketch for that part. Hence, no synchronization mechanism is required since all these constructions are independent from each other. The results of this set of experiments are given in Table I.

As the results in Table I show, merged tabulation increases the performance of the CMS construction even with extra overhead of managing an auxiliary array, i.e., `hashes`$[.]$ in Figure 4. Instead of using a single register as in naive tabulation shown in Figure 2, in merged tabulation, the intermediate hash values are first written to the auxiliary memory. Once all d hashes are computed, they are read back from the memory again. However, these extra reads as well as the hash table accesses are more cache friendly.

The impact of merged tabulation in terms of improvement percentage behaves differently on **Arch-1** and **Arch-2** and for **small** and **large** sketches. On **Arch-2**, the improvements on the CMS construction time with $\tau = 1$, 2 and 4 threads are 10.5%, 11% and 12% on **small** sketches and 6.5%, 7% and 8% on **large** sketches, respectively. Hence, the amount of the improvement does not change with the number of threads and sketch sizes. On **Arch-1**, the improvement starts with 4.5% with a single thread for the **small** sketch, merged tabulation increases the runtime by 2% for two threads., with $\tau = 2$, 4, 8 and 16, the improvement becomes 20%, 46%, 51% and

			Naive Tabulation					Merged Tabulation				
			τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	64.3	33.1	23.1	14.0	7.6	60.4	34.4	17.5	10.0	5.2
		large	72.3	36.8	20.0	9.8	4.9	60.6	33.3	17.1	8.8	4.5
	Arch-2	**small**	31.8	16.0	8.0			28.3	14.2	7.1		
		large	33.2	16.7	8.4			30.8	15.4	7.8		
Zipf 0.9	**Arch-1**	**small**	63.9	40.8	25.4	14.4	7.5	61.1	34.0	17.4	9.5	5.1
		large	67.6	34.1	19.2	9.5	4.9	60.4	33.7	17.3	8.9	4.4
	Arch-2	**small**	31.6	15.9	8.2			28.6	14.3	7.3		
		large	32.8	16.5	8.3			30.8	15.4	7.7		
Zipf 1.1	**Arch-1**	**small**	65.8	32.2	19.6	11.9	6.1	60.0	33.1	16.6	9.0	4.8
		large	65.7	33.2	18.4	9.4	4.6	60.0	32.5	16.8	8.4	4.2
	Arch-2	**small**	31.5	15.9	8.4			28.8	14.6	7.5		
		large	32.4	16.1	10.1			29.6	15.0	8.9		
Zipf 1.5	**Arch-1**	**small**	63.2	32.7	17.8	10.6	5.7	59.7	30.7	15.4	7.9	4.0
		large	63.8	32.0	16.3	8.2	4.1	59.7	30.3	15.3	7.6	3.8
	Arch-2	**small**	31.1	15.7	8.0			28.6	14.4	7.4		
		large	30.9	15.5	7.8			28.3	14.3	7.2		
Zipf 2.0	**Arch-1**	**small**	67.8	32.8	17.1	9.1	4.3	59.8	30.3	15.1	7.7	3.9
		large	63.0	31.6	16.0	7.9	4.0	59.8	30.0	15.0	7.5	3.7
	Arch-2	**small**	31.1	15.6	8.5			28.8	14.8	8.0		
		large	31.1	15.5	9.5			28.2	14.3	8.9		
Zipf 3.0	**Arch-1**	**small**	62.9	31.9	16.4	8.3	4.2	59.8	29.9	14.9	7.5	3.7
		large	63.0	31.5	15.7	7.9	3.9	60.0	29.9	15.0	7.5	3.7
	Arch-2	**small**	31.0	15.8	8.3			28.7	14.6	7.4		
		large	31.2	15.5	9.8			28.4	14.4	8.9		

TABLE I: Parallel CMS construction times (in seconds) with naive and merged tabulation for $\tau \in \{1, 2, 4, 8, 16\}$ threads on **Arch-1** and $\tau \in \{1, 2, 4\}$ threads on **Arch-2**. The stream sizes for **Arch-1** and **Arch-2** are $N = 2^{30}$ and $N = 2^{25}$, respectively. The rows labeled with **small** and **large** denote the 8×211 and 8×2003 CMS tables, respectively.

47%, respectively. On the other hand, for the **large** sketch, the improvement is more stable and in between 2% and 12% for varying number of threads.

The results show that a smaller sketch on **Arch-1** can benefit from the merged tabulation better. We believe that this is due to the common cache lines among the last row and the first row of two partial sketches. With naive tabulation, a thread accesses to a partial sketch after each hash computation. Hence, the accesses are frequent with short intervals in between. On the other hand, with merged tabulation, each d consecutive accesses are combined, and in between, d hash functions are computed. This reduces the probability of concurrent updates on the same cache line by two threads.

B. Parallel CMS construction with a single sketch

Although using partial sketches is pleasingly parallelizable and can be practical for some use cases, it may not a practical approach especially for memory restricted devices such as Raspberry Pi. The additional memory might be required by other applications running on the same hardware and/or other types of skeches being maintained at the same time for the same or a different data stream. This is why in the rest of the experiments, we focus on the single CMS sketch case. Similarly, we apply merged tabulation for all the implementation in the rest of the paper.

Table II shows the results of the experiments in which a single sketch is constructed in parallel. As expected, the naive parallel approach suffer from false sharing with multiple threads on **Arch-1**. However, we did not observe the such a dramatic slowdown on **Arch-2**. Furthermore, the improvement behaves differently on these two architectures. Since buffering the hashes has an overhead, we expect a slowdown in a sequential execution. This is what we observe on **Arch-1**; the runtimes are increased to 68.9 and 68.6 seconds from 61.4 and 60.4 seconds, respectively for **small** and **large** sketches. However, we do not see the same pattern on **Arch-2**. We believe this is due to the existence of a relatively smaller L1 cache on **Arch-2** which is equipped with a 16KB L1 cache per core and a 512KB L2 cache per processor. On **Arch-1**, we have 32KB L1, 256KB L2 caches per core and 30MB L3 cache per processor. In the naive parallel approach, locations in the merged hash table and in the sketch are accessed one after another in an alternating fashion. Hence, these structures need to live in the cache at the same time for efficiency. An advantage of the buffered parallel approach is that it separates the accesses to these structures. Hence, it does not use the cache for these structures at the same time.

Overall, the buffered parallel approach is significantly better than the naive parallel one; it improves the runtimes and yields a better scalability. However, for the **small** table on

Arch-1, the proposed approach yields much less parallel efficiency compared to the results of the **large** sketch. This is expected since, with multiple threads performing simultaneous updates on the sketch, whose rows are stored consecutively on the memory, false sharing can reduce the performance. Furthermore, the impact will be more drastic for smaller w values which is the case in our experiments. To avoid false sharing as much as possible, we added extra padding to each row of CMS sketch. The third set of columns in Table II shows the execution times after padding. As the results confirm, padding helps to get rid of the problems arose from increased coherency traffic due to the concurrent updates on the same cache line by multiple threads.

Figure 5 summarizes all the experiments on parallel construction of a single CMS sketch with individual execution times for the first and the second stages of buffered parallel executions. An interesting observation is that for almost of all executions, the relative execution time of Stage 2 to whole execution time is much larger for **Arch-2**, i.e., 91% to 78% for single core.

C. Other optimizations

In Algorithm 3, there is an implicit barrier between the first and second stages of the buffered parallel construction. This barrier guarantees that the buffer is filled and ready to be processed. For our experiments, we removed this barrier and implemented the buffered algorithms with OpenMPs `nowait` directive. Although this may have a negative impact on the accuracy, when b is large, the probability of a thread using a buffer entry which is not correctly set is small. Hence, we opt to remove the barrier. As Table III shows, this has a positive impact on the performance. To remove the threat to validity of the experiments, we measured how much we lost from the accuracy. Figure 6 show the differences on the estimations for multithreaded executions compared to a single thread execution. The most frequent 1000 items are used for both charts in the figure. As the charts show, the differences are negligible considering that the real frequencies are in the order of tens of thousands for both cases. etween

VI. Related Work

CMS is proposed by Cormode and Muthukrishnan structure to summarize data streams [11] and it quickly became popular. They also suggest on sequential, parallel and hardware implementations of CMS [10] where the first parallel algorithm distributes rows to the threads so each thread computes the corresponding hash value. However, the hash computations are not merged as in our study. Another suggested algorithm is using multiple CMSs from different sub-streams and merge the results later.

Cafaro et al. propose an augmented frequency sketch for time-faded heavy hitters: recent items are considered to be more important [6]. The authors extend this work via parallelization by dividing the stream into sub-streams and generating multiple sketches instead of a single one [5]. When a query is issued, the result is computed using a reduce operation (as we did with multiple CMS tables in this study). There are studies in the literature employing synchronization primitives such as atomic operations for frequency counting [13]. To the best of our knowledge, our work is the first cache-friendly, synchronization-free, single-table CMS algorithm specifically designed for limited-memory multicore architectures such as Raspberry Pi or other single-board computers. Furthermore, it utilizes tabular hashing which is recently shown to provide good statistical properties and reported to be fast [12], [23]. When multiple hashes on the same item are required, which is the case for many sketches, our table-merging technique will be useful for algorithms using tabular hashing.

CMS has also been used as an underlying structure to design more advanced sketches. Recently, Roy et al. developed ASketch which filters high frequent items first and handles the remaining with a sketch such as CMS which they used for implementation [21]. However, their parallelization depends on SPMD which stores multiple filters/sketches. Another advanced sketch employing multiple CMSs for parallelization is [22]. If a single CMS sketch is neccesary, e.g., to fit into the cache, our techniques can be employed to develop more advanced parallel sketches using table-based ones such as CMS or Count Sketch.

VII. Conclusion

In this work, we investigated parallelization of Count-Min Sketch on two multicore architectures; a high-end server and a single board computer. We proposed two main techniques. The first one, merged tabulation, is useful when a single is item needs to be hashed multiple times and can be used for different sketches and other applications performing the same task. The second technique is a classical one in high-performance computing, where the intermediate results are buffered to correctly synchronize the computation and regularize the memory accesses. The experiments we performed show that both techniques work for CMS construction on multicore architectures.

As a future work, we are planning to analyze the configuration options of the processors on single board computers such as how much data/instruction cache they use and how they handle coherency. Since sketches already incur inaccuracies, a freedom to have a little bit more can yield various optimizations. We also want to extend the architecture spectrum with other single board computers such as Odroid with heterogeneous cores and Parallela having many cores. We believe that similar techniques we develop here can also be used for other sketches.

References

[1] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: ACM, 1996, pp. 20–29. [Online]. Available: http://doi.acm.org/10.1145/237814.237823

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

			Naive Parallel (Algorithm 2)					Buffered Parallel (Algorithm 3)					Buffered Parallel with Padding (Algorithm 3 + Padding)				
			τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	61.5	299.9	304.4	201.7	156.9	68.3	60.9	45.5	30.4	28.0	68.6	43.8	23.5	13.5	11.1
		large	60.1	233.8	229.2	139.2	96.5	68.7	51.4	29.8	17.1	14.4	68.8	44.0	23.7	13.6	11.1
	Arch-2	**small**	28.5	15.4	7.8			27.5	13.8	6.9			27.6	13.8	6.9		
		large	30.8	15.4	7.7			26.8	13.3	6.6			26.8	13.3	6.6		
Zipf 0.9	**Arch-1**	**small**	61.4	289.8	287.0	176.6	118.3	68.0	61.9	45.0	29.7	27.2	68.3	46.7	24.4	13.6	10.9
		large	60.4	240.0	225.3	137.2	96.9	68.6	50.3	28.2	15.8	13.4	68.5	43.6	22.7	12.7	10.4
	Arch-2	**small**	28.7	15.4	7.7			27.7	13.8	7.4			27.8	13.8	7.5		
		large	30.7	15.4	7.7			26.8	13.3	8.3			27.2	13.3	8.3		
Zipf 1.1	**Arch-1**	**small**	60.3	287.6	287.3	179.3	125.9	68.1	58.4	40.1	26.2	24.1	68.09	43.5	22.9	13.0	10.6
		large	60.9	146.1	222.2	159.1	130.5	68.3	51.4	28.7	16.0	13.2	68.4	46.4	24.3	13.6	10.8
	Arch-2	**small**	28.7	15.5	8.6			27.7	13.9	7.39			27.9	13.9	7.4		
		large	30.2	15.4	9.5			26.8	13.3	8.1			27.1	13.3	8.1		
Zipf 1.5	**Arch-1**	**small**	62.8	331.8	314.2	206.8	166.1	68.0	53.0	34.2	20.6	18.5	68.1	44.3	24.2	13.8	11.4
		large	63.4	214.2	257.5	193.6	170.5	68.3	47.5	25.8	14.6	12.1	68.3	44.2	23.6	13.6	11.0
	Arch-2	**small**	30.7	16.4	8.5			27.4	13.7	6.9			27.3	13.7	7.0		
		large	30.8	16.1	10.3			26.5	13.2	7.4			26.5	13.2	7.5		
Zipf 2.0	**Arch-1**	**small**	62.8	340.5	330.0	236.8	207.2	67.7	49.0	27.6	15.8	14.0	67.7	44.5	23.6	13.1	10.6
		large	64.6	261.7	311.0	232.3	204.3	68.0	44.5	23.6	14.9	12.7	68.0	43.7	23.1	13.7	10.5
	Arch-2	**small**	31.2	16.5	9.5			27.3	13.7	7.2			27.4	13.7	7.2		
		large	31.2	16.3	10.5			26.2	13.0	7.9			26.4	13.0	7.9		
Zipf 3.0	**Arch-1**	**small**	62.9	372.5	340.8	278.5	250.8	67.6	45.2	24.6	13.8	11.4	67.7	43.1	22.9	12.8	10.5
		large	62.9	319.2	349.0	282.1	253.5	67.7	38.6	22.6	13.0	11.0	68.1	44.4	23.1	13.1	11.2
	Arch-2	**small**	30.7	16.4	8.5			27.0	13.5	7.2			27.3	13.7	7.2		
		large	30.8	16.1	10.3			26.2	13.0	7.4			26.3	13.1	7.4		

TABLE II: Parallel CMS construction times (in seconds) with a single sketch for $\tau \in \{1, 2, 4, 8, 16\}$ threads on **Arch-1** and $\tau \in \{1, 2, 4\}$ threads on **Arch-2**. The stream sizes for **Arch-1** and **Arch-2** are $N = 2^{30}$ and $N = 2^{25}$, respectively. The rows labeled with **small** and **large** denote the 8×211 and 8×2003 CMS tables, respectively. The **Naive Parallel** construction does not apply buffering or padding whereas the next two sets of columns apply these techniques. For **Buffered Parallel** construction w/out padding, when $\tau = 16$, all the threads are utilized for the first stage of Algorithm 3, i.e., hashing, but only $d = 8$ of them are used for the second stage.

[3] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[4] A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM J. Comput.*, vol. 28, no. 5, pp. 1627–1640, May 1999. [Online]. Available: https://doi.org/10.1137/S0097539795294165

[5] M. Cafaro, M. Pulimeno, and I. Epicoco, "Parallel mining of time-faded heavy hitters," *Expert Systems with Applications*, vol. 96, pp. 115 – 128, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417417307777

[6] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Mining frequent items in the time fading model," *Information Sciences*, vol. 370-371, pp. 221 – 238, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025516305618

[7] B. Chandramouli, M. Ali, J. Goldstein, B. Sezgin, and B. S. Raman, "Data stream management systems for computational finance," *Computer*, vol. 43, no. 12, pp. 45–52, Dec 2010.

[8] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ser. ICALP '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 693–703. [Online]. Available: http://dl.acm.org/citation.cfm?id=646255.684566

[9] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 464–475. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315492

[10] G. Cormode and M. Muthukrishnan, "Approximating data with the count-min sketch," *IEEE Softw.*, vol. 29, no. 1, pp. 64–69, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/MS.2011.127

[11] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58 – 75, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677403001913

[12] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, "Practical hash functions for similarity estimation and dimensionality reduction," in *NIPS*, 2017.

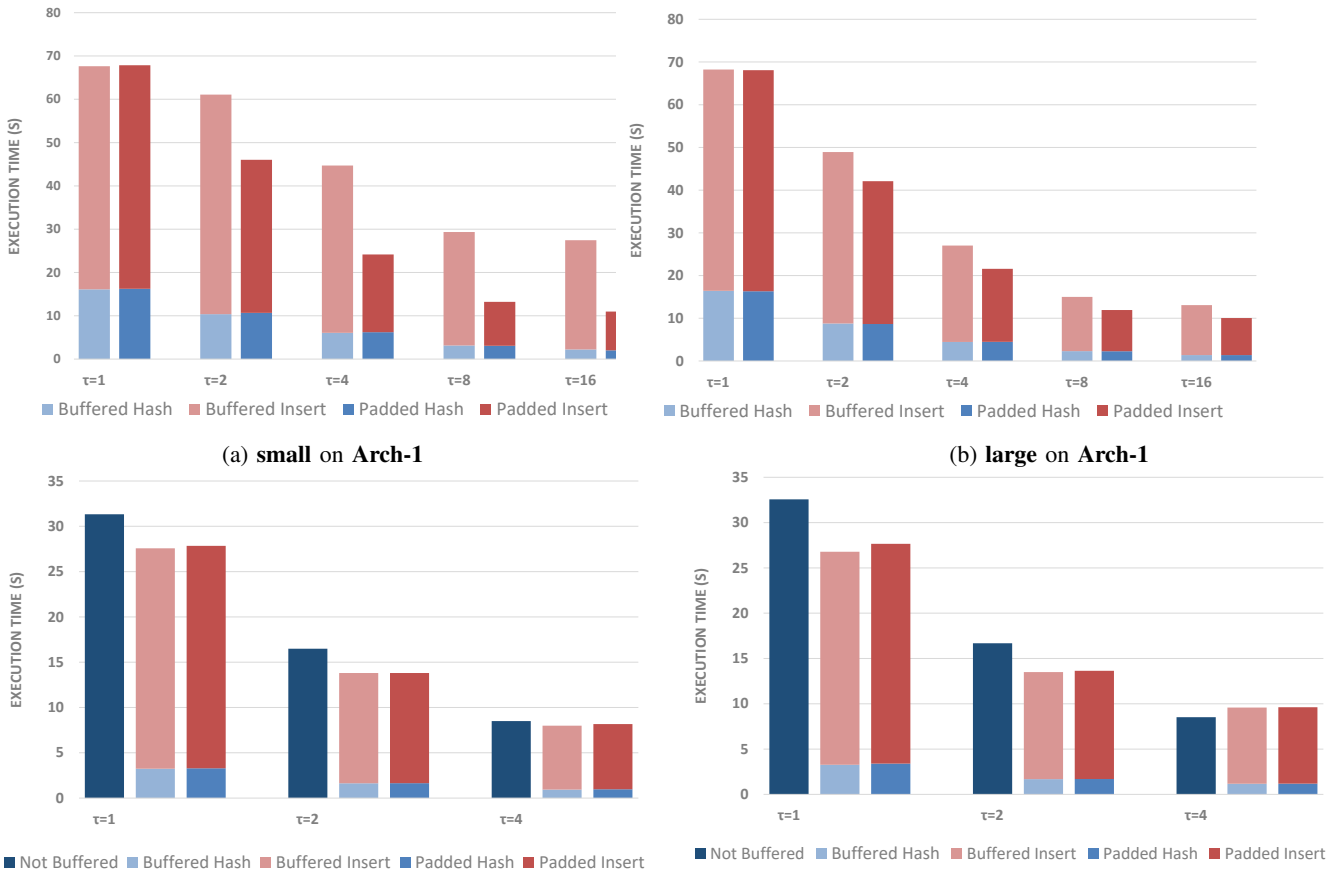
[13] S. Das, S. Antony, D. Agrawal, and A. El Abbadi, "Thread cooperation in multicore architectures for frequency counting over multiple data streams," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 217–228, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687653

[14] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/564691.564699

[15] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 280–292, Jun. 2001. [Online]. Available: http://dx.doi.org/10.1109/90.929851

[16] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '98. New York, NY, USA: ACM, 1998, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/285237.285287

[17] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "How to summarize the universe: Dynamic maintenance of quantiles," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 454–465. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287409



(a) **small** on **Arch-1**

(b) **large** on **Arch-1**

(c) **small** on **Arch-2**

(d) **large** on **Arch-2**

Fig. 5: Execution times (in seconds) for hashing and update/insertion stages for single CMS sketch experiments. The execution times for the naive parallel approach are omitted for **Arch-1** since they are relatively much larger and deteriorate the visibility of the improvements.

[18] A. Goyal, H. Daumé, III, and G. Cormode, "Sketch algorithms for estimating point queries in nlp," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, ser. EMNLP-CoNLL '12. Stroudsburg, PA, USA: Association for Computational Linguistics, 2012, pp. 1093–1103. [Online]. Available: http://dl.acm.org/citation.cfm?id=2390948.2391070

[19] S. Muthukrishnan, "Data streams: Algorithms and applications," *Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, pp. 117–236, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1561/0400000002

[20] M. Pătraşcu and M. Thorup, "The power of simple tabulation hashing," *J. ACM*, vol. 59, no. 3, pp. 14:1–14:50, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2220357.2220361

[21] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 1449–1463. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882948

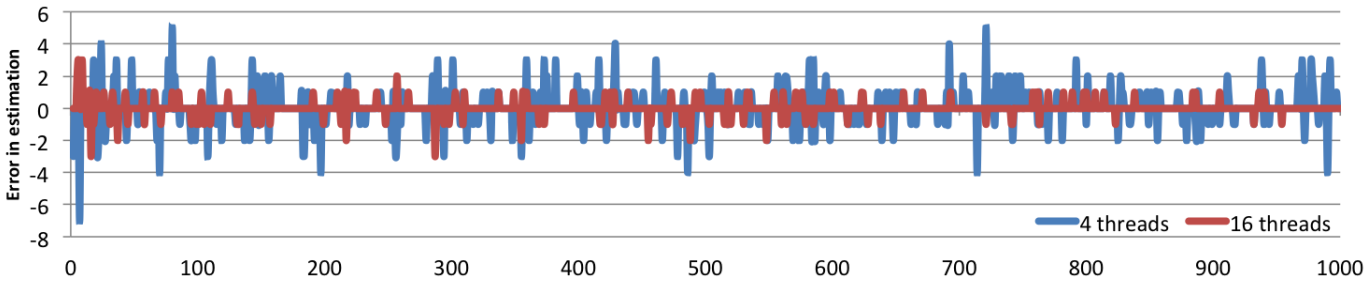
[22] D. Thomas, R. Bordawekar, C. Aggarwal, and P. S. Yu, "A Frequency-aware Parallel Algorithm for Counting Stream Items on Multicore Processors," IBM, Tech. Rep., 2007.

[23] M. Thorup, "Fast and powerful hashing using tabulation," *Commun. ACM*, vol. 60, no. 7, pp. 94–101, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3068772

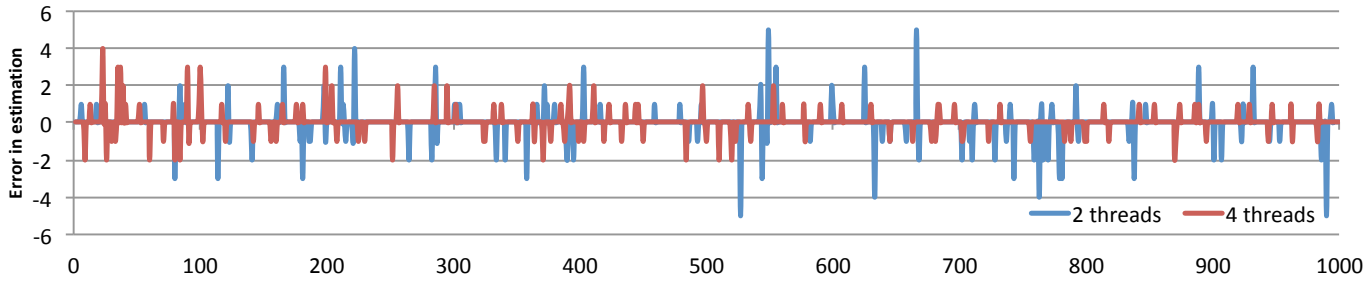
[24] A. L. Zobrist, "A new hashing method with application for game playing," *Technical Report 88*, University of Wisconsin, Madison, Wisconsin, 1970.

			Buffered Parallel with Padding					Buffered Parallel with Padding + Barrier				
			τ=1	τ=2	τ=4	τ=8	τ=16	τ=1	τ=2	τ=4	τ=8	τ=16
Uniform	**Arch-1**	**small**	68.7	43.5	22.8	13.0	10.6	68.6	43.8	23.5	13.5	11.1
		large	68.7	45.0	23.2	13.0	10.6	68.8	44.0	23.7	13.6	11.1
	Arch-2	**small**	28.3	13.9	8.8			28.7	14.3	10.4		
		large	27.4	13.39	8.5			27.8	13.9	9.6		
Zipf 0.9	**Arch-1**	**small**	68.3	46.7	24.4	13.6	10.9	69.0	44.7	23.8	13.7	11.2
		large	68.5	43.6	22.7	12.7	10.4	68.7	44.7	24.2	13.8	11.3
	Arch-2	**small**	27.8	13.8	7.5			27.9	13.8	8.0		
		large	27.2	13.3	8.3			27.6	13.6	9.3		
Zipf 1.1	**Arch-1**	**small**	68.1	43.5	22.9	13.0	10.6	68.2	45.6	24.6	14.2	11.5
		large	68.4	46.4	24.3	13.6	10.8	68.3	44.7	24.1	13.8	11.3
	Arch-2	**small**	27.8	13.9	7.4			27.9	13.8	7.8		
		large	27.1	13.3	8.1			27.5	13.5	9.3		
Zipf 1.5	**Arch-1**	**small**	67.9	44.6	23.5	13.1	10.6	68.1	44.3	24.2	13.8	11.4
		large	68.0	43.7	23.1	13.0	10.5	68.3	44.2	23.6	13.6	11.0
	Arch-2	**small**	27.3	13.7	7.0			27.3	13.7	7.0		
		large	26.5	13.2	7.5			26.4	13.1	7.49		
Zipf 2.0	**Arch-1**	**small**	67.7	44.5	23.6	13.1	10.6	68.0	45.3	24.5	14.0	11.4
		large	68.0	43.7	23.1	13.7	10.5	67.9	47.5	26.5	15.5	12.1
	Arch-2	**small**	27.4	13.7	7.29			27.6	13.7	8.1		
		large	26.2	13.0	7.9			26.2	13.0	7.8		
Zipf 3.0	**Arch-1**	**small**	67.7	43.1	22.9	12.8	10.5	67.8	45.6	24.3	13.9	11.4
		large	68.1	44.4	23.1	13.1	11.2	67.9	44.8	23.9	13.7	11.2
	Arch-2	**small**	27.3	13.7	7.2			27.6	13.7	8.2		
		large	26.3	13.1	7.4			26.2	13.1	7.8		

TABLE III: Impact of removing the barrier between the stages of buffered parallel execution on performance. The first set of the columns is the same with the last set of columns in Table II. The values in the table are in seconds.



(a) **large** on **Arch-1**



(b) **small** on **Arch-2**

Fig. 6: The difference between the frequency estimations of single and multithreaded executions of buffered parallel approach without the barrier. The charts show the differences for the top 1000 items for the **large** sketch on **Arch-1** and **small** sketch on **Arch-2**.