Hash-Grams On Many-Cores and Skewed Distributions

Edward Raff
Laboratory for Physical Sciences
edraff@lps.umd.edu
Booz Allen Hamilton
raff_edward@bah.com

Mark McLean
Laboratory for Physical Sciences
mrmclea@lps.umd.edu

***Abstract*—When using n-grams for features, it is often the case that an expedient and effective first-pass of feature selection can be performed by picking the top-k most frequent features. The hash-gram approach was introduced as a method of quickly performing this feature selection. In this work we identify a failure case of parallelizing the hash-gram algorithm to a large number of CPU cores P when the data is highly skewed. We resolve this issue to produce a hash-gram algorithm with consistent performance across potential skewness-es and number of CPU cores, making it practically usable for big-data cases where more powerful compute is needed.**

***Index Terms*—n-grams, feature selection, parallelization**

I. Introduction

In many predictive applications, keeping track of the exact input features is not necessary to obtain good predictive accuracy. Approaches such as the hashing-trick have exploited this to obtain practical benefits at the cost of losing this direct understanding of the features.

Recently *hash-grams* [1] were proposed as one such method as an alternative to the hashing-trick over n-grams when many irrelevant features are present. The hash-gram approach was developed because the number of uninformative features per malware datum dwarfed the number of relevant ones, making the hashing-trick ineffective. Hash-grams were also shown to provide a 10x-788x speedup compared to the classical Space-Saving algorithm, which is another approach to select the top-k most frequent items in a stream. These properties make the hash-gram algorithm attractive for big-data applications which may be too computationally demanding with prior approaches.

While the hash-gram algorithm was shown to obtain decent speedups on a machine with 6 CPU cores, it also showed that the scaling of the hash-gram approach was sensitive to the skewness of the input distribution. This is problematic when attempting to use a many-core machine, limiting the scalability of the approach. By further investigating this relationship, we show that the hash-gram algorithm's sensitivity to data skew prevents it from scaling to larger numbers of CPU cores, and can even cause performance regressions as the number of CPUs P continues to increase.

To make hash-grams relevant to big-data applications where processing with a larger number of CPUs P is necessary, we introduce a simple modification that practically eliminates the performance sensitivity with respect to data skew. This is achieved introducing a thread local write-buffer for memory access, when the write-buffer is normally a technique for disk IO. We show that significant gains in scalability are achieved on a machine with up to 80 CPU cores.

The remainder of our paper is organized as follows. First we will discuss the alternative approaches to the hash-gram algorithm in section II. In section III we will discussed the hash-gram approach and show how it can fail to achieve meaningful speedups depending on the data skewness in. Then we will detail in section IV our write-buffer approach, and analyze how it should remediate the scaling issue with high confidence. Then we will show empirical results in section V that our improved hash-gram with write buffer provides consistent speedups across a range of potential skewness, out-performs the Space-Saving algorithm for parallel computation, and reduces runtime on a real-world malware dataset. Finally, we will conclude in section VII.

II. Related Work

The most directly related approach to hash-grams is a method known as the hashing-trick [2], [3]. The hashing-trick maps features to indices in a sparse feature vector, and the index is selected via a hash-function of the input feature. Once the index is selected, the index is incremented (or decremented, depending on the value returned by the hash) by the original coefficient. Because the index of every item is determined by the hash value, no memory is needed to keep a map from tokens to index.

The usefulness of the hashing-trick is dependent on selecting a target feature space size D', such that most values will remain zero and collisions will be infrequent. However, the hash-gram approach was developed for cases where the number of features per data point is on the order of 2 million, meaning D' would need to be on the order of 20 million to obtain just 10% non-zeros. Beyond a computational requirement in RAM, this makes the learning problem harder due to the curse of dimensionality [4].

Because only a smaller set of one hundred thousand to one-million features are frequent enough for use, the hashing trick is wasteful and slow for our use case. Instead we would like to determine the top-k most frequent features in the first place, reducing overall computation.

A number of algorithms have been developed to select the top-k most frequent items from a running stream of data using a fixed amount of memory with only one pass over the data [5]–[9]. By treating the features from documents as a stream, we can use these methods to select the top-k most frequent items of the stream. Compared to the hash-gram approach, this has the advantage of retaining the original objects — which aids in model interoperability.

The Space-Saving algorithm [10] in particular has become one of the preferred algorithms for top-k selection from a stream. In cases like our own, where only insertions into the index are performed, its been found that the Space-Saving algorithm is especially effective with a number of advantages in terms of guarantees on the error rate, higher recall and precision, and computational efficiency [11]. A parallelized version of the Space-Saving algorithm has also been developed by running multiple instances independently, and then merging the results after [12], [13]. As such it makes an ideal comparison point in terms of efficiency for selecting the top-k. A primary difference in intended uses exists though. The Space-Saving algorithm is often used with the desire to keep $k \leq 10,000$ items. In our case we want to select the top $k = 1,000,000$ features. This is a larger target population than the space-saving algorithm has historically been used for.

III. Hash-Grams and Parallelism

Hash-Grams where developed as an alternative to feature selection when working with n-grams, where the occurrence of individual n-grams often follow a power-law distribution such as the Zipfian distribution. In this scenario, only a small subset of features will occur frequently enough to be informative to a machine learning classifier. The majority of features will likely occur only a few times, making them uninformative. Tracking n-gram counts to determine the top-k most frequent n-grams can be computationally demanding, but is a simple and effective first-pass to feature selection [14]. For example, Raff, Zak, Cox, *et al.* [15] found that selecting the top 100,000 most frequent byte 6-grams was a more effective feature selection process than seven other first-order feature selection approaches (e.g., information-gain).

For large alphabets, such as when using large n-gram sizes, it is not practical to track every individual n-gram occurrence. Doing so requires out-of-core processing that immediately moves the computational bottleneck to disk-IO systems, or event to network-IO if running in a distributed fashion. Hash-grams solve this problem by keeping track of only the top-k *hashes* of the features, rather than the original features themselves.

The algorithm specifying the hash-gram approach is given in Algorithm 1, and in summary has the following steps:

1) Create a large table T of size B to store hashes in
2) hash ($h(\cdot)$) each item and increment the index of the hash in the table
3) return the top-k hashes by count based on the index from the hash table.

Algorithm 1 Hash-Graming [1]

Require: Bucket size B, rolling hash function $h(\cdot)$, corpus of S documents, and desired number of frequent hash-grams k.

```
T ← new integer array of size B
for all documents x ∈ S do                    ▷ Done in parallel
    for n-gram g ∈ x do
        q' ← h(q) mod B
        T[q'] ← T[q'] + 1                     ▷ Update atomically
T_k ← QuickSelet(T, k)                        ▷ O(B)
return T_k
```

The approach is simple and allows for scaling the size of the dataset considerably compared to exact counting or other approximate counting approaches. While noise in the features occurs since collisions will occur in the hash-table, collisions come from intrinsically rare features and so do not significantly impede the accuracy of trained models. Raff and Nicholas [1] also showed that if the input distribution follows the Zipfian distribution, then there is a high probability that all top-k hashes corresponding to the true top-k items will be obtained.

A. Parallel Scaling Issues

The hash-gram algorithm was argued to be easy to parallelize by simply processing documents in parallel, and performing the update to T using atomic operations. This is indeed easy to implement, but does not work efficiently for all Zipfian distributions, especially when many CPU cores are available.

To understand this, we will review in more detail the Zipfian distribution characterized by an alphabet of N possible characters and shape parameter ϱ, which has the Probability Mass Function (PMF) given in Equation 1. $H_N^{(\varrho)} = \sum_{i=1}^{N} 1/i^{\varrho}$ indicates the N'th harmonic number of the ϱ'th order.

$$f(x; \varrho, N) = \begin{cases} \frac{x^{-\varrho-1}}{H_N^{(\varrho+1)}} & 1 \leq x \leq N \\ 0 & \text{else} \end{cases} \tag{1}$$

The cumulative distribution function (CDF) $F(x; \varrho, N)$ is given by Equation 2

$$F(x; \varrho, N) = \begin{cases} \frac{H_x^{(\varrho+1)}}{H_N^{(\varrho+1)}} & 1 \leq x \leq N \\ 1 & x > N \end{cases} \tag{2}$$

The Zipfian distribution is one type of power-law distribution, in which the item of rank 1 is the most frequent, each successive rank has significantly reduced probability of occurring. As $\varrho \to 0$, the distribution becomes progressively flatter. As $\varrho \to \infty$, the probability mass of the Zipfian begins to concentrate almost entirely on the first few ranks.

We take a moment to clarify that the parameter ϱ does not directly translate to the skewness of the resulting Zipfian distribution, though many prior works refer to it as the skewness parameter [5], [11], [13]. While the probability mass concentrates onto about the first rank item as $\varrho \to \infty$, the skewness as defined by the third standardized moment of the

Zipfian distribution is not monotonic with ϱ. Thus for more precise clarity, when we refer to skewness for the rest of this work we are referring to ϱ causing the majority (e.g., $\geq 95\%$) of probability mass to concentrate on fewer and fewer items.

The issue with the hash-gram approach occurs with respect to the shape parameter ϱ of the distribution. As $\varrho \to \infty$, the PDF of (1) will begin to collapse, as we have just discussed. In the extreme case, almost every sample will return the same token (i.e., $x = 1$ — the rank 1 item).

When running in a single-threaded mode ($P = 1$), this behavior begins to provide speed advantages. Despite the large table T in Algorithm 1, the same value in the table ($x = 1$) will be accessed over and over again, keeping its value in L1 cache and resulting in faster updates.

As multiple threads are used, contention via the atomic updates will begin to increase for frequently accessed indices. The greater the skew, the more the updates will focus on a smaller set of values, and the greater the contention will be.

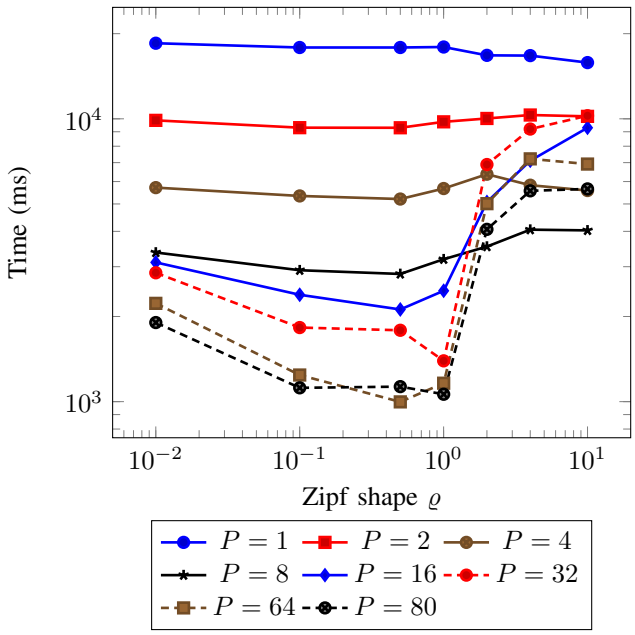

Fig. 1: Runtime (in milliseconds, y-axis) of the hash-gram algorithm on synthetic data from the Zipfian distribution $f(x; \varrho, 2^{31} - 1)$. The x-axis shows the shape parameter ϱ of the distribution, and each plot shows the results at a different level of parallelism P.

Both of these behaviors can be seen in Figure 1, where we show the total runtime to insert 10^8 samples from the Zipfian distribution into the hash-gram table. The server used for this experiment, and all others in this paper, has four Intel Xeon CPUs at 2.1 GHz, each with 20 CPU cores for a total of 80 cores. All experiments will be run with hyper-threading disabled, and the process pinned to cores across the minimum number of CPUs.

When the data distribution is flatter (e.g., $\varrho = 0.1$), we see that increasing the number of CPU cores regularly increases the speedup of the algorithm, with diminishing returns starting at around $P = 16$ CPU cores.

The story changes as the skewness ϱ increases. When $P = 1$, we see a slow but significant reduction in runtime as ϱ increases due to the aforementioned caching advantages. This benefit disappears immediately at $P = 2$ with the impact of update contention, and we instead see a minor increase in runtime for $P = 2$ as ϱ goes to 10. The degradation starts to become significant at $P = 8$ CPU cores, and by $P = 32$ cores the speedup is no-better than if we had used only 2 CPUs.

We further note the importance that we see significant performance regressions occur at $P = 16$ CPU cores. This means that the issue is not caused by inter-CPU communication over the slower the QPI bus, which has a latency two-orders of magnitude slower than intra-CPU communication [16]. Instead just the MESIF protocol that is used to ensure cache-coherence between cores is enough to induce these dramatic slow downs [17].

This means that while the hash-gram approach will return the top-k hashes with high-probability, the algorithm does not scale well as many scores P are introduced for $\varrho > 1$. This is problematic as datasets with greater asymetry in the PMF are those for which the algorithm's theory best supports its use, but practically can not scale as well. Now that we have identified this issue, we seek to remove this roadblock so that the hash-gram strategy can be applied more freely without worrying about just how lop-sided the input distribution may be.

IV. Hash-Grams with a Write Buffer

The naive approach to solving this issue would be to have each thread create its own local copy of the global array T, and then merge the counts of all T items at the end. This may seem intuitive because not all B entries of T need to be merged. Only the top $k \cdot P$ items from each worker need to be communicated to guarantee that the exact same top-k items are retrieved. However, this means $P \cdot B$ memory is needed for the P tables. Since B is routinely a value on the order of 2^{32}, this explodes memory cost from the 10 GB scale to the 1 TB scale.

To resolve the performance regressions of the hash-gram approach in a memory efficient manner, we introduce a thread-local write-buffer to each individual thread. While write-buffers are not a new concept, they are usually used to improve disk based write throughput and generally involve one write buffer which multiple threads access [18]. Instead we leverage them to improve memory IO-throughput to reduce contention, and thus increase parallel efficiency. Our use of the write-buffers is also different from how they are classically used to reduce disk-IO. Under normal circumstances, the write-buffer is used to accumulate a larger number of writes to perform together at a later point in time to increase efficiency. In our case we will be using smaller buffers and using a simple eviction strategy to perform writes from the buffer one-at-a-time. The details of our approach will be explained below.

Given a thread context P_i, we will keep track of a write-buffer WB^{P_i} that tracks the number of times a given hash has been incremented. The size of this write buffer will be

small, containing $\mathcal{O}(P)$ entries for each thread-local buffer. For this reason we also keep track of the original hash value that occupies the local write-buffer in an "identity" buffer ID^{P_i}.

Our new hash-grams with write-buffer approach can then be summarized as follows, with more explicit detail given in Algorithm 2:

1) Initiate hash table as normal hash-gram, including thread local write buffers
2) For each feature, compute its hash and find its position in the local buffer. If something occupies its space, evict the occupant.
3) Increment the count for the hash in the local buffer.
4) When all items in a thread's processing queue are done, evict all items from local buffer.

Algorithm 2 Hash-Graming with Write-Buffer

Require: Bucket size B, rolling hash function $h(\cdot)$, corpus of S documents, and desired number of frequent hash-grams k, and total threads P.

$T \leftarrow$ new integer array of size B
$\text{WB}^{P_i} \leftarrow$ new thread-local integer array of size P for each of the $i \in [1, P]$ threads.
$\text{ID}^{P_i} \leftarrow$ new thread-local integer array of size P for each of the $i \in [1, P]$ threads.
for all documents $x \in S$ **do** ▷ *Done in parallel over P processors*
  for n-gram $g \in x$ **do**
    $q' \leftarrow h(q) \mod B$
    $q'' \leftarrow q' \mod P$
    if $\text{ID}^{P_i}[q''] = q'$ **then** ▷ *This hash is already in write buffer*
      $\text{WB}^{P_i}[q''] \leftarrow \text{WB}^{P_i}[q''] + 1$
    else ▷ *Evict old value and update buffer*
      $q_{\text{old}} \leftarrow \text{ID}^{P_i}[q'']$
      $\delta \leftarrow \text{WB}^{P_i}[q'']$
      $T[q_{\text{old}}] \leftarrow T[q_{\text{old}}] + \delta$ ▷ *Update atomically*
      $\text{WB}^{P_i}[q''] \leftarrow 1$
      $\text{ID}^{P_i}[q''] \leftarrow q'$
$T_k \leftarrow \text{QuickSelet}(T, k)$
return T_k

Because each thread context will get its own local buffer with only $\mathcal{O}(P)$ memory requirements, the additional memory cost of our approach is minuscule. In particular we will use $P \cdot c$ total capacity for each local buffer, and will use a value of $c = 5$ for all experiments unless specified otherwise. This small size means it is easy to keep the entire write buffer in local L1 or L2 cache.

Since each thread is maintaining its own local write buffer, we do not need to implement any additional thread-safety or locking mechanism, keeping overhead minimal. To further minimize overhead, we do not use any First-In First-Out or Least Recently Used mechanism. Instead items will be removed from the write buffer based on collisions with new entries.

The simplicity of this buffer means that the overhead for updating the local write buffer with an eviction costs are on the order of an L1 memory access read and write. The evicted item must then perform the more expensive index into main memory to perform an atomic update to the shared T array. This later step was the only step before. Since the access to L1 cache is orders of magnitude faster than main-memory updates, the relative cost of our buffer will be negligible in the case that $\varrho \leq 1$ (i.e., the case of low contention from Figure 1 will continue to perform well).

When contention is higher ($\varrho > 1$), the efficiency of our approach is dependent on the local buffers working to collect multiple updates to frequent items before evicting them due to a hash collision. We perform such analysis below to show that we expect this approach to reduce contention despite the lack of any new communication logic between threads and buffers.

A. Analytic Analysis of Write Buffer Efficiency

In analyzing the benefit of the write buffer, we will divide the inputs coming into the hash-gram algorithm into two groups: *hot* items and *infrequent* items. Hot items are those which are seen frequently and cause atomic update contention due to their frequency. Infrequent items are those that do not cause any communication overhead because they do not incur CAS contention. For our analysis we assume that only the top κ items from the Zipf distribution are "hot", and causing contention via CAS updates. Necessarily, $\kappa < P$, as if there where more "hot" items than threads, there would be little contention to perform updates because each thread could be updating a different hot item. Then all threads would be performing updates and no contention would occur.

Because all updates first hit the local write buffer WB, there are only a select number of scenarios:

1) An infrequent item evicts another infrequent item from the table: In this case only minor overhead occurs to perform a read and write from the local buffer, which due to its small size should be contained within L1/L2 cache and have fast access.
2) A hot item evicts an infrequent item: The overhead is still small, and the hot item should begin to collect larger counts.
3) A hot item evicts another hot item: This scenario is rare due to the limited number of hot items.
4) An infrequent item evicts a hot item: So long as this occurs infrequently, we obtain lowered contention by collecting the increments locally and coalescing the updates into one.

Scenarios 1 and 2 are of little consequence, as they involve the eviction of an infrequent item from the buffer. We necessarily expect this to happen continuously due to the large number of individually infrequent items. As such we are interested scenarios 3 and 4.

Hot Evicts Hot: For scenario 3, we are concerned that hot items may evict other hot items. Because items are placed into the local buffer based on their hash value, if both items are hot, we then expect the issue to occur continuously and cause update contention with other threads. As such we want to know the

probability that two hot items will share the same local buffer position. This amounts to the well known Birthday Problem, where the number of potential "birthdays" is the local table size and the number of "people" is the number of hot items.

Looking back at Figure 1, we can see the problem with skewness causing contention occurs once $\varrho > 1$. If we use $\varrho = 2$, then at least 95% of all probability mass will collapse onto the first three items $\forall N \geq 4$, so essentially we would have only $\kappa = 3$ hot items. Using $P \cdot 5$ buckets on our 80-core server, this means there is a 99.25% chance that all items will belong to differing buckets.

We can calibrate the multiplier c used on the number of processors P to larger values if we desire higher confidence that no hot-hot birthday collisions occur. Even a multiplier of 100 would only place us in the KB range for the local buffers, making the cost minimal. Values of $\varrho \leq 1$ do not need to be considered because they violate our $\kappa < P$ assumption and thus have no performance issues, as is seen in Figure 1. Thus we do not have to worry about scenario 3.

Infrequent Evicts Hot: Scenario four is that an infrequent item evicts a hot item. We expect this to occur regularly, the goal is that the rate of occurrence will be lower, such that only 1 out of P threads will be performing a CAS update of the hot item at a time. If only one thread is performing a CAS update, then the update can proceed with no contention overhead. To simplify our analysis, we assume that all hot items are currently in the local buffer WB. This is likely to occur on a regular basis since the hot items occur regularly. We want to know the probability that the next item will be an infrequent item that evicts a hot item. If this probability is less than $1/P$, than intuitively we obtain a situation where we expect no contention for updates on average. This is because we will have P threads each performing an update to T with probability $1/P$.

When a new item $g \sim \text{Zipf}(N, \varrho)$ arrives, the probability that it is one of κ hot items is the CDF of the Zipf distribution, $F(\kappa; N, \varrho)$. In this case the local buffer for g gets incremented, and no updates to the global count array T are performed. Then with a probability $1 - F(\kappa; N, \varrho)$ we will obtain an item g that is an infrequent item. Once an infrequent item is sampled, its also necessary that it collided with one of the κ hot items to cause an eviction we care about. If the infrequent item collided with another infrequent item, that would devolve back to scenario 1 which we already know to be of low overhead.

The sampling of an infrequent item and it having the same hash index in the local write buffer as a hot item are independent events. Thus we arrive at a the probability of the next item g causing an eviction of a hot item is $(1 - F(\kappa; N, \varrho)) \cdot \kappa / P$.

Using our previous example of $\varrho = 2$ and $\kappa = 3$ hot items that account for 95% of the probability mass, our 80 core machines would have a probability of only 0.1875%. This is less than the 1.25% probability for the $1/P$ goal, and is only reduced by the fact that we use $P \cdot 5$ in our experiments.

We note as well that this analysis is somewhat pessimistic, as we are assuming that the eviction of any hot item should occur with probability $\leq 1/P$. In actuality, if two different hot items are evicted from differing buffers at the same time, they will not cause any increased contention because they will go to different indexes in the table T.

As such we have now shown analytically how we expect our local write buffer approach to reduce update contention of hot items in the global count table T.

V. Experiments

Now that we have described our new hash-gram approach with write buffers and analytically shown how it reduces communication overhead, we will provide empirical experiments validating the results. For all experiments we will be inserting 10^8 samples from the Zipfian distribution so that the distribution of samples is known and we can look at the change in performance as a function of ϱ. We continue to use our four socket Intel Xeon CPUs at 2.1 GHz, each with 20 CPU cores for a total of 80 cores. All experiments will be run with hyper-threading disabled, and the process pinned to cores across the minimum number of CPUs.

A. Write Buffers Empirically Improve Performance

We first look at results under the same experiment shown in Figure 1, where we look at the time to digest all samples as a function of ϱ. The results using our new write buffer approach are shown in Figure 2.

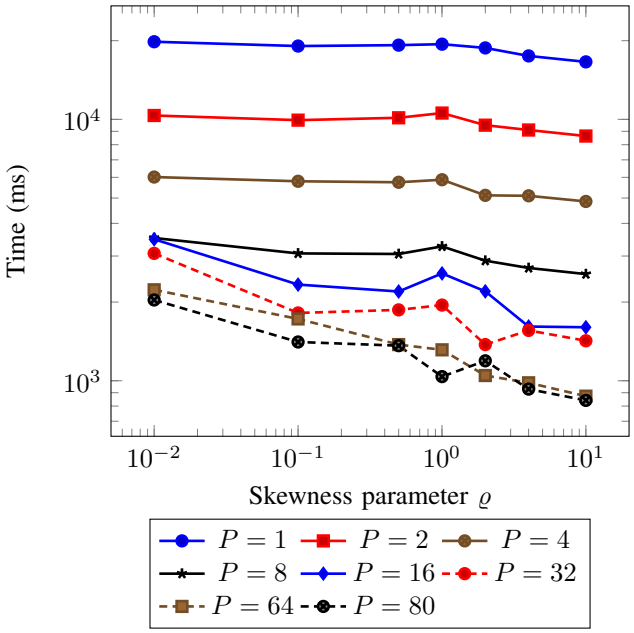

Fig. 2: Runtime (in milliseconds, y-axis) of the hash-gram *with buffer* algorithm on synthetic data from the Zipfian distribution $f(x; \varrho, 2^{31} - 1)$. The x-axis shows the skewness parameter ϱ of the distribution, and each plot shows the results at a different level of parallelism.

Originally we saw that for the naive hash-gram approach, runtime decreased as ϱ increased when $P = 1$, but began to degrade as more cores where used and $\varrho > 1$, creating a bisection of behavior types. We now see that for $P \in [1, 80]$, no dramatic performance regressions occur. The observation that

runtime decreases as ϱ increases now applies across all numbers of threads, and actually becomes stronger as P increases.

Related to this behavior, we can see runtime characteristics improve with the number of cores and ϱ. While there are no performance regressions, we continue to observe diminished returns in speedup after $P = 16$ CPU cores when $\varrho \leq 1$, which appears to be caused by the memory-wall phenomena [19]. When the input distribution is sufficiently uniform, most updates require accessing an index in T that was not previously in cache, and thus requires a trip to main-memory. The total amount of bandwidth to main-memory is limited, and appears to be saturating as more cores are used. However, as $\varrho > 1$, we obtain cache efficiencies for the hot items that re-occur regularly and make the majority of the probability mass. This gets leveraged by the local write buffers for fast updates, and reduces the demand on the amount of needed memory bandwidth.

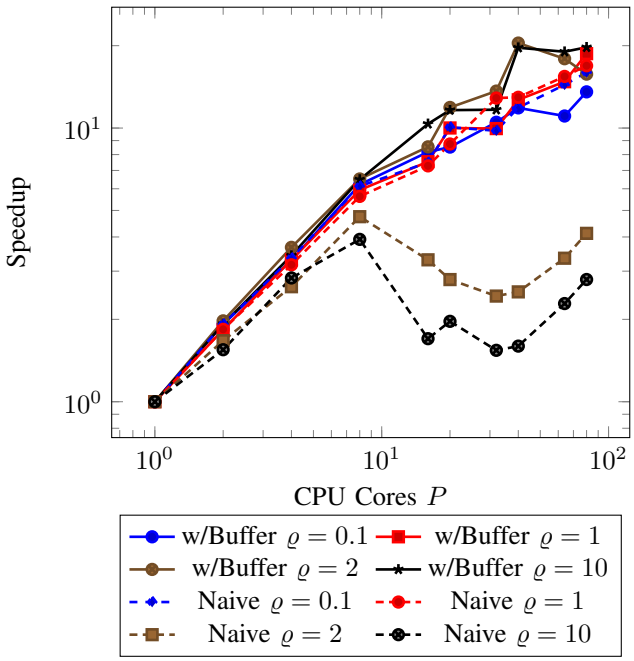

Fig. 3: Runtime (in milliseconds, y-axis) of the hash-gram *with buffer* algorithm on synthetic data from the Zipfian distribution $f(x; \varrho, 2^{31} - 1)$. The x-axis shows the skewness parameter ϱ of the distribution, and each plot shows the results at a different level of parallelism.

We show overall speedup results for different values of ϱ in Figure 3. Here we make note that the speedup for hash-grams is measured relative to the single-core performance using the same value of ϱ. This is necessary to accurately capture the differences due to cache benefits exhibited after we introduce our write buffer. In the plot, solid lines show the performance with our new write buffers, and dashed lines show the naive original hash-gram approach to multi-core parallelism. Here it becomes clear that the hash-gram with buffer has consistent speedup, with moderate improvement as ϱ increases — where the naive approach has initial speed improvements but declines into significant performance regressions.

B. *Hash-Grams vs Space-Saving*

We now look at the relative performance difference between the hash-gram data structure and the parallel space-saving algorithm. The parallel variant was introduced by Cafaro, Pulimeno, and Tempesta [12], who showed near linear speedups on Zipfian data when keeping track of the top $k = 4000$ items for up to $P = 8$ cores. Later Cafaro, Pulimeno, Epicoco, *et al.* [13] showed significant speedups when using up to $k = 8,000$ items on up to $P = 512$ CPUs via MPI programming. In doing so they observed that parallel efficiency did between to decreased as k increased, and we use a Java implementation of the Space-Saving algorithm[1] to create a level playing field with our Java hash-gram implementation.

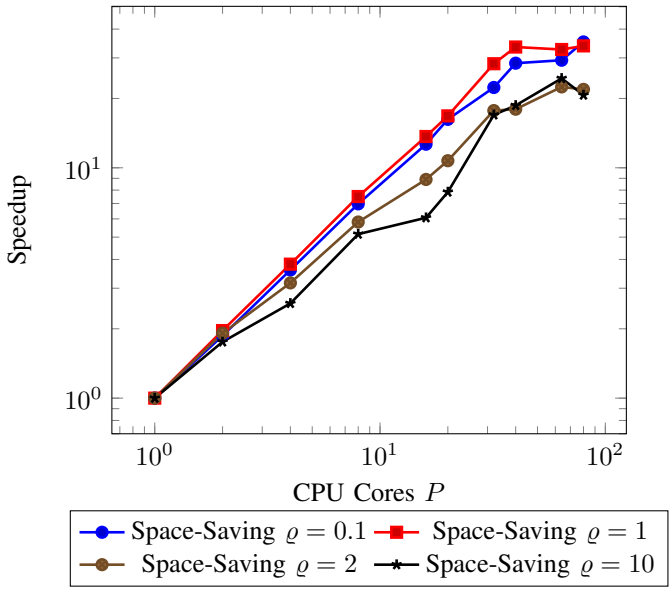

Fig. 4: Speedup of the Space-Saving algorithm on synthetic data from the Zipfian distribution $f(x; \varrho, 2^{31} - 1)$. The x-axis shows the number of CPU cores P used, and each plot shows the results at a different level of parallelism. Done with $B = 10,000$.

In Figure 4 we look at the speedup of just the Space-Saving algorithm as the shape parameter ϱ changes from one extreme (0.1) to another (10) when seeking to obtain the top $B = 10,000$ elements. In this case our results confirm those reported by Cafaro, Pulimeno, Epicoco, *et al.* [13] for a similar value of k over a wider range of ϱ. However, we can not look at the speedup of Space-Saving in isolation, because the standard single-core hash-gram algorithm is faster than the space-saving approach. We are also interested in a much larger value of B than has been investigated in prior works of the Space-Saving algorithm.

In Figure 5 we show the relative speed of the Space-Saving algorithm compared to the hash-gram approach on the same data with the same number of cores in use. This is done with a more realistic value of having the Space-Saving algorithm

[1]Implementation from https://github.com/fzakaria/space-saving

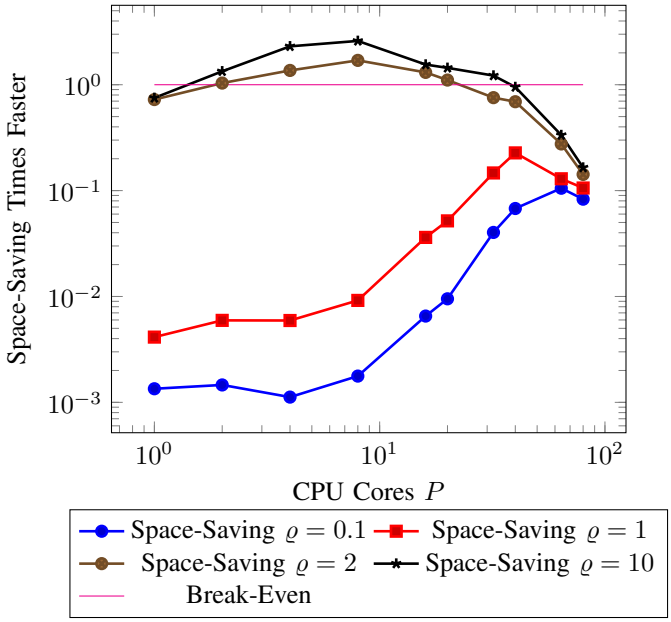

Fig. 5: The y-axis shows the relative Speed of the Space-Saving algorithm compared to that of the hash-gram approach. Values greater than one (shown by the magenta line) indicate that Space-Saving is faster, and below one that hash-grams are faster. The x-axis shows the number of CPU cores P used, and each plot shows the results at a different level of parallelism. Done with $B = 200,000$.

track $B = 200,000$ items, so that we can hopefully obtain an accurate estimate of the top $k = 100,000$ items.

In doing so we see that the Space-Saving algorithm is competitive in performance when $\varrho > 1$ and $P \leq 32$. While slightly slower in initial single-threaded performance, the improved parallel scaling of the Space-Saving algorithm allows it to become computationally faster than the hash-gram approach by a maximum factor of 2.6. However, as P increases the communication overhead of the parallel space-saving begins to dominate and make it over 12 times slower than the hash-gram approach.

When the Zipfian distribution has a relatively flatter distribution ($\varrho \leq 1$), we see that the additional overhead of the Space-Saving updates makes it 100 to 1000 times slower than the hash-gram approach for only 1 CPU core. This does not begin to improve until after $P = 16$ CPU cores, where the speedup of the Space-Saving algorithm out-paces that of the hash-gram approach. However this ends quickly at $P \geq 32$, where again the communication overhead of the parallel Space-Saving algorithm begins to dominate costs. Overall, the Space-Saving algorithm is always 10-100x slower than the hash-gram approach for low-skewed data.

At best, the Space-Saving algorithm has comparable and slightly better runtime for a limited number of cores P and only for $\varrho > 1$. Overall the hash-gram with our write buffer provides consistently high throughput across all combinations of ϱ and P. In additional the Space-Saving algorithm has the disadvantage that it was only tracking $B = 200,000$ items total. Practical use is further hampered by not knowing how much larger B must be compared to k for the Space-Saving algorithm to return the exact top-k items. Under the Zipfian assumption, worse case proven bounds would require $B = \mathcal{O}(k^2)$, where the hash-grams probabilistic proof keeps the cost linear with k so long as $k < B$. This was shown in the original hash-gram paper [1]. This means we could require considerably larger values of B for the Space-Saving algorithms, and performance decreased as the value of B increases [1]. The hash-gram's approach performance is minimally impacted by the desired number of items k, and with our write-buffer approach, parallelizes well across a wide range of ϱ and P.

VI. Malware Data Experiment

All of the prior experimentations and tests in this section have been focused on data generated from the Zipfian distribution. This allows us to quantify and isolate the root cause of the performance issue. We now look at performance on a real-world malware classification dataset.

The data we use comes from [20], which has approximately 2-million binaries evenly divided between benign and malicious. Each file is g-zip compressed, and occupies 3.5 TB of disk space when compressed. Computing 6-byte n-grams as features for this dataset originally took 2 weeks on a cluster with 12 servers.

Following the original hash-gram paper, we have implemented the hash-gram algorithm in Java and tested it on this same corpus, and use the Rabin-Karp rolling hash function [21] to convert byte sequences into hashed integers.

Apriori we would not expect our write-buffer improvements to have as dramatic an impact on this dataset. When sampling data from the Zipfian distribution the atomic contention and updates make the majority of work. On this real world data a number of other items require CPU time but do parallelize well, namely reading the files from disk and decompressing them in memory. As such Amdahl's law [22] tells us we should expect a reduced impact from improving parallel efficiency. At the same time, our experimentation tells us contention overhead will be significant when data is highly skewed, which prior work has found to be true for byte n-grams of malware [15].

When we run the original hash-gram algorithm on this corpus, we find that the total runtime is 12 hours and 36 minutes. Using our improved hash-gram with write-buffer, we see execution time drop to 11 hours and 32 minutes. This gives us a 9% improvement in runtime on a real world dataset where the hash-gram updates of the global array T are the minority of the work involved, showing that the issue we identify is of practical concern and can be resolved with our write-buffer approach. This speed improvement comes with no appreciable change in model accuracy, which is in line with prior work using the hashing-trick that obtained almost no quality degradation even with a 40% collision rate [2].

VII. Conclusion

We have studied a performance deficiency of the hash-gram algorithm when a large number of processors P are to be used.

When data is of high skewness, atomic overhead contention dramatically reduces scaling. We have resolved this problem by introducing a write-buffer approach on in-memory updates to a larger table. In doing so we observe performance becomes consistent and better behaved across a wide spectrum of CPU cores $P \in [2, 80]$ and data skewness distributions $\varrho \in [0.01, 10]$. This allows the hash-gram approach to provide speedups by up to 100x compared to the Space-Saving algorithm for the many-core scenario.

References

[1] E. Raff and C. Nicholas, "Hash-Grams: Faster N-Gram Features for Classification and Malware Detection," in *Proceedings of the ACM Symposium on Document Engineering 2018*, 2018. DOI: 10.1145/3209280.3229085.

[2] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and V. Vishwanathan, "Hash kernels," in *International Conference on Artificial Intelligence and Statistics*, 2009, pp. 496–503.

[3] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, 2009, pp. 1113–1120. DOI: 10.1145/1553374.1553516.

[4] P. Indyk and R. Motwani, "Approximate Nearest Neighbors : Towards Removing the Curse of Dimensionality 1 Introduction," *Young*, 1999.

[5] G. Cormode and S. Muthukrishnan, "Summarizing and mining skewed data streams," in *Proc. of the 2005 SIAM International Conference on Data Mining*, 2005, pp. 44–55. DOI: 10.1137/1.9781611972757.5.

[6] 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. DOI: 10.1016/j.jalgor.2003.12.001.

[7] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy, "Tracking Join and Self-join Sizes in Limited Storage," in *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '99, 1999, pp. 10–20. DOI: 10.1145/303976.303978.

[8] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01, 2001, pp. 79–88.

[9] 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, 2002, pp. 693–703.

[10] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams," in *Proceedings of the 10th International Conference on Database Theory*, ser. ICDT'05, 2005, pp. 398–412. DOI: 10.1007/978-3-540-30570-5{_}27.

[11] G. Cormode and M. Hadjieleftheriou, "Methods for Finding Frequent Items in Data Streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010. DOI: 10.1007/s00778-009-0172-z.

[12] M. Cafaro, M. Pulimeno, and P. Tempesta, "A parallel space saving algorithm for frequent items and the Hurwitz zeta distribution," *Information Sciences*, vol. 329, pp. 1–19, 2016. DOI: 10.1016/J.INS.2015.09.003.

[13] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Parallel Space Saving on Multi and Many-Core Processors," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 7, 2018. DOI: 10.1002/cpe.4160.

[14] H. P. Luhn, "The Automatic Creation of Literature Abstracts," *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 159–165, 1958. DOI: 10.1147/rd.22.0159.

[15] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, 2016. DOI: 10.1007/s11416-016-0283-1.

[16] D. Molka, D. Hackenberg, R. Schone, and W. E. Nagel, "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture," in *2015 44th International Conference on Parallel Processing*, 2015, pp. 739–748. DOI: 10.1109/ICPP.2015.83.

[17] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," *Resource*, vol. 3, no. 2, 2011.

[18] J. Cieslewicz, K. A. Ross, and I. Giannakakis, "Parallel Buffers for Chip Multiprocessors," in *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, ser. DaMoN '07, 2007, 2:1–2:10. DOI: 10.1145/1363189.1363192.

[19] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04, 2004, pp. 162–167. DOI: 10.1145/977091.977115.

[20] E. Raff and C. Nicholas, "Malware Classification and Class Imbalance via Stochastic Hashed LZJD," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17, 2017, pp. 111–120. DOI: 10.1145/3128572.3140446.

[21] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987. DOI: 10.1147/rd.312.0249.

[22] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.