Fast and scalable minimal perfect hashing for massive key sets

Antoine Limasset, Guillaume Rizk, Rayan Chikhi, Pierre Peterlongo

Fast and scalable minimal perfect hashing for massive key sets

Antoine Limasset[1], Guillaume Rizk[1], Rayan Chikhi[2], and Pierre Peterlongo[1]

1 **IRISA Inria Rennes Bretagne Atlantique, GenScale team, Campus de Beaulieu 35042 Rennes, France**
2 **CNRS, CRIStAL, Université de Lille, Inria Lille - Nord Europe, France**

Abstract

Minimal perfect hash functions provide space-efficient and collision-free hashing on static sets. Existing algorithms and implementations that build such functions have practical limitations on the number of input elements they can process, due to high construction time, RAM or external memory usage. We revisit a simple algorithm and show that it is highly competitive with the state of the art, especially in terms of construction time and memory usage. We provide a parallel C++ implementation called *BBhash*. It is capable of creating a minimal perfect hash function of 10^{10} elements in less than 7 minutes using 8 threads and 5 GB of memory, and the resulting function uses 3.7 bits/element. To the best of our knowledge, this is also the first implementation that has been successfully tested on an input of cardinality 10^{12}. Source code: https://github.com/rizkg/BBHash

1998 ACM Subject Classification H.3.1 E.2

Keywords and phrases Minimal Perfect Hash Functions, Algorithms, Data Structures, Big Data

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.11

1 Introduction

Given a set S of N elements (*keys*), a minimal perfect hash function (`MPHF`) is an injective function that maps each key of S to an integer in the interval $[1, N]$. In other words, an `MPHF` labels each key of S with integers in a collision-free manner, using the smallest possible integer range. A remarkable property is the small space in which these functions can be stored: only a couple of bits per key, independently of the size of the keys. Furthermore, an `MPHF` query is done in constant time. While an `MPHF` could be easily obtained using a key-value store (e.g. a hash table), such a representation would occupy an unreasonable amount of space, with both the keys and the integer labels stored explicitly.

The theoretical minimum amount of space needed to represent an `MPHF` is known to be $\log_2(e)N \approx 1.44N$ bits [10, 14]. In practice, for large key sets (billions of keys), many implementations achieve less than $3N$ bits per key, regardless of the number of keys [2, 9]. However no implementation comes asymptotically close to the lower bound for large key sets. Given that `MPHF`s are typically used to index huge sets of strings, e.g. in bioinformatics [6, 7, 8], in network applications [12], or in databases [5], lowering the representation space is of interest. We observe that in many of these applications, `MPHF`s are actually used to construct static dictionaries, i.e. key-value stores where the set of keys is fixed and never updated [6, 8]. Assuming that the user only queries the `MPHF` to get values corresponding to keys that are guaranteed to be in the static set, the keys themselves do not necessarily need to be stored in memory. However the associated values in the dictionary typically do need to be stored, and

they often dwarf the size of the MPHF. The representation of such dictionaries then consists of two components: a space-efficient MPHF, and a relatively more space-expensive set of values. In such applications, whether the MPHF occupies 1.44 bits or 3 bits per key is thus arguably not a critical aspect.

In practice, a significant bottleneck for large-scale applications is the construction step of MPHFs, both in terms of memory usage and computation time. Constructing MPHFs efficiently is an active area of research. Many recent MPHF construction algorithms are based on efficient peeling of hypergraphs [1, 3, 4, 11]. However, they require an order of magnitude more space during construction than for the resulting data structure. For billions of keys, while the MPHF itself can easily fit in main memory of a commodity computer, its construction algorithm requires large-memory servers. To address this, Botelho and colleagues [4] propose to divide the problem by building many smaller MPHFs, while Belazzougui *et al.* [1] propose an external-memory algorithm for hypergraph peeling. Very recently, Genuzio *et al.* [11] demonstrated practical improvements to the Gaussian elimination technique, that make it competitive with [1] in terms of construction time, lookup time and space of the final structure. These techniques are, to the best of our knowledge, the most scalable solutions available. However, when evaluating existing implementations, the construction of MPHFs for sets that significantly exceed a billion keys remains prohibitive in terms of time and space usage.

A simple idea has been explored by previous works [6, 12, 16] for constructing PHFs (Perfect Hash Functions, non minimal) or MPHFs using arrays of bits, or fingerprints. However, it has received relatively less attention compared to other hypergraph-based methods, and no implementation is publicly available in a stand-alone MPHF library. In this article we revisit this idea, and introduce novel contributions: a careful analysis of space usage during construction, and an efficient, parallel implementation along with an extensive evaluation with respect to the state of the art. We show that it is possible to construct an MPHF using almost as little memory as the space required by the final structure, without partitioning the input. We propose a novel implementation called *BBhash* ("Basic Binary representAtion of Successive Hashing") with the following features:

- construction space overhead is small compared to the space occupied by the MPHF,
- multi-threaded,
- scales up to to very large key sets (tested with up to 1 trillion keys).

To the best of our knowledge, there does not exist another usable implementation that satisfies any two of the features above. Furthermore, the algorithm enables a time/memory trade-off: faster construction and faster query times can be obtained at the expense of a few more bits per element in the final structure and during construction. We created an MPHF for ten billion keys in 6 minutes 47 seconds and less than 5 GB of working memory, and an MPHF for a trillion keys in less than 36 hours and 637 GB memory. Overall, with respect to other available MPHF construction approaches, our implementation is at least two orders of magnitudes more space-efficient when considering internal and external memory usage during construction, and at least one order of magnitude faster. The resulting MPHF has slightly higher space usage and faster or comparable query times than other methods.

2 Efficient construction of minimal perfect hash function

2.1 Method overview

Our MPHF construction procedure revisits previously published techniques [6, 12]. Given a set F_0 of keys, a classical hash function h_0 maps keys to an integer in $[1, |F_0|]$. A bit array A_0 of

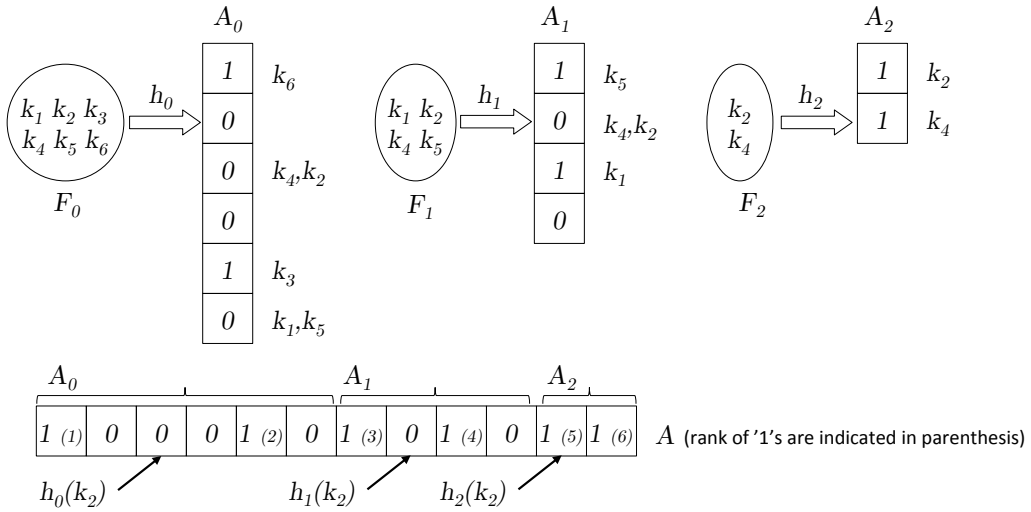

Figure 1 `MPHF` construction and query example. The input is a set F_0 composed of $N = 6$ keys (k_1 to k_6). All keys are hashed using a hash function h_0 and are attempted to be placed in an array A_0 at positions given by the hash function. The keys k_3 and k_6 do not have collisions in the array, thus the corresponding bits in A_0 are set to '1'. The other keys from F_0 that are involved in collisions are placed in a new set F_1. In the second level, keys from F_1 are hashed using a hash function h_1. Keys k_1 and k_5 are uniquely placed while k_2 and k_4 collide, thus they are then stored in the set F_2. With the hash function h_2, the keys from F_2 have no collision, and the process finishes. The `MPHF` query operation is very similar to the construction algorithm. Let A be the the concatenation of A_0, A_1, A_2 (see bottom part of the figure). To query k_2, the key is first hashed with h_0. The associated value in A_0 is '0', so k_2 is then hashed with h_1. The value associated in A_1 is again '0'. When finally hashed with h_2, the value associated in A_2 is '1' and thus the query stops here. The index returned by the `MPHF` is the rank of this '1' (here, 5) in A. In this example, the `MPHF` values returned when querying k_1, k_2, k_3, k_4, k_5 and k_6 are respectively 4,5,2,6,3, and 1.

size $|F_0|$ is created such that there is a 1 at position i if and only if exactly one element of F_0 has a hash value of i. We say that there is a *collision* whenever two keys in F_0 have the same hash value. Keys from F_0 that were involved in a collision are inserted into a new set F_1. The process repeats with F_1 and a new hash function h_1. A new bit array A_1 of size $|F_1|$ is created using the same procedure as for A_0 (except that F_1 is used instead of F_0, and h_1 instead of h_0). The process is repeated with $F_2, F_3, \ldots$ until one of these sets, $F_{\text{last}+1}$, is empty.

We obtain an `MPHF` by concatenating the bit arrays $A_0, A_1, \ldots, A_{\text{last}}$ into an array A. To perform a query, a key is hashed successively with hash functions $h_0, h_1, \ldots$ as long as the value in A_i ($i \geq 0$) at the position given by the hash function h_i is 0. Eventually, by construction, we reach a 1 at some position of A for some $i = d$. We say that the *level* of the key is d. The index returned by the `MPHF` is the rank of this one in A. See Figure 1 for an example.

2.2 Algorithm details

2.2.1 Collision detection

During construction at each level d, collisions are detected using a temporary bit array C_d of size $|A_d|$. Initially all C_d bits are set to '0'. A bit of $C_d[i]$ is set to '1' if two or more keys from

F_d have the same value i given by hash function h_d. Finally, if $C_d[i] = 1$, then $A_d[i] = 0$. Formally:

$$C_d[i] = 1 \Rightarrow A_d[i] = 0;$$
$$(h_d[x] = i \text{ and } A_d[i] = 0 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 1 \,(\text{and } C_d[i] = 0)\,;$$
$$(h_d[x] = i \text{ and } A_d[i] = 1 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 0 \text{ and } C_d[i] = 1.$$

2.2.2 Queries

A query of a key x is performed by finding the smallest d such that $A_d[h_d(x)] = 1$. The (non minimal) hash value of x is then $(\sum_{i<d} |F_i|) + h_d(x)$.

2.2.3 Minimality

To ensure that the image range of the function is $[1, |F_0|]$, we compute the cumulative rank of each '1' in the bit arrays A_i. Suppose, that d is the smallest value such that $A_d[h_d(x)] = 1$. The minimal perfect hash value is given by $\sum_{i<d}(weight(A_i) + rank(A_d[h_d(x)])$, where $weight(A_i)$ is the number of bits set to '1' in the A_i array, and $rank(A_d[y])$ is the number of bits set to 1 in A_d within the interval $[0, y]$, thus $rank(A_d[y]) = \sum_{j<y} A_d[j]$. This is a classic method also used in other MPHFs [3].

2.2.4 Faster query and construction times (parameter γ)

The running time of the construction depends on the number of collisions on the A_d arrays, at each level d. One way to reduce the number of collisions, hence to place more keys at each level, is to use bit arrays (A_d and C_d) larger than $|F_d|$. We introduce a parameter $\gamma \in \mathbb{R}$, $\gamma \geq 1$, such that $|C_d| = |A_d| = \gamma|F_d|$. With $\gamma = 1$, the size of A is minimal. With $\gamma \geq 2$, the number of collisions is significantly decreased and thus construction and query times are reduced, at the cost of a larger `MPHF` structure size. The influence of γ is discussed in more detail in the following analyses and results.

2.3 Analysis

Proofs of the following observations and lemma are given in the Appendix.

2.3.1 Size of the `MPHF`

The expected size of the structure can be determined using a simple argument, previously made in [6]. When $\gamma = 1$, the expected number of keys which do not collide at level d is $|A_d|e^{-1}$, thus $|A_d| = |A_{d-1}|(1 - e^{-1}) = |A_0|(1 - e^{-1})^d$. In total, the expected number of bits required by the hashing scheme is $\sum_{d\geq 0} |A_d| = N \sum_{d\geq 0}(1 - e^{-1})^d = eN$, with N being the total number of input keys ($N = |F_0|$). Note that consequently the image of the hash function is also in $[1, eN]$, before minimization using the rank technique. When $\gamma \geq 1$, the expected proportion of keys without collisions at each level d is $|A_d|e^{-\frac{1}{\gamma}}$. Since each A_d no longer uses one bit per key but γ bits per key, the expected total number of bits required by the `MPHF` is $\gamma e^{\frac{1}{\gamma}} N$.

2.3.2 Space usage during construction

We analyze the disk space used during construction. Recall that during construction of level d, a bit array C_d of size $|A_d|$ is used to record collisions. Note that the C_d array is only

needed during the d-th level. It is deleted before level $d+1$. The total memory required during level d is $\sum_{i\leq d}(|A_i|)+|C_d| = \sum_{i<d}(|A_i|)+2|A_d|$.

▶ Lemma 1. For $\gamma > 0$, the space of our MPHF is $S = \gamma e^{\frac{1}{\gamma}} N$ bits. The maximal space during construction is S when $\gamma \leq \log(2)^{-1}$, and $2S$ bits otherwise.

A full proof of the Lemma is provided in the Appendix.

3 Implementation

We present *BBhash*, a C++ implementation available at http://github.com/rizkg/BBHash. We describe in this section some design key choices and optimizations.

3.1 Rank structure

We use a classical technique to implement the rank operation: the ranks of a fraction of the '1's present in A are recorded, and the ranks in-between are computed dynamically using the recorded ranks as checkpoints.

In practice 64 bit integers are used for counters, which is enough for realistic use of an MPHF, and placed every 512 positions by default. These values were chosen as they offer a good speed/memory trade-off, increasing the size of the MPHF by a factor 1.125 while achieving good query performance. The total size of the MPHF is thus $(1+\frac{64}{512})\gamma e^{\frac{1}{\gamma}} N$.

3.2 Parallelization

Parallelization is achieved by partitioning keys over several threads. The algorithm presented in Section 2 is executed on multiple threads concurrently, over the same memory space. Built-in compiler functions (e.g. *sync_fetch_and_or*) are used for concurrent access in the A_i arrays. The efficiency of this parallelization scheme is shown in the Results section, but note that it is fundamentally limited by random memory accesses to the A_i arrays which incur cache misses.

3.3 Hash functions

The MPHF construction requires classical hash functions. Other authors have observed that common hash functions behave practically as well as fully random hash functions [2]. We therefore choose to use xor-shift based hash functions [13] for their efficiency both in terms of computation speed and distribution uniformity [15].

3.4 Disk usage

In the applications we consider, key sets are typically too big to fit in RAM. Thus we propose to read them on the fly from disk. There are mainly two distinct strategies regarding the disk usage during construction: 1/ during each level d, keys that are to be inserted in the set F_{d+1} are written directly to disk. The set F_{d+1} is then read during level $d+1$ and erased before level $d+2$; or 2/ at each level all keys from the original input key file are read and queried in order to determine which keys were already assigned to a level $i<d$, and which would belong to F_d. When the key set becomes small enough (below user-defined threshold) it is loaded in ram to avoid costly re-computation from scratch at each level.

The first strategy obviously provides faster construction at the cost of temporary disk usage. At each level $d>0$, two temporary key files are stored on disk: F_d and F_{d+1}. The highest disk

usage is thus achieved during level 1, i.e. by storing $|F_1|+|F_2| = |F_0|((1-e^{-1/\gamma})+(1-e^{-1/\gamma})^2)$ elements. With $\gamma = 1$, this represents $\approx 1.03N$ elements, thus the construction overhead on disk is approximately the size of the input key file. Note that with $\gamma = 2$ (resp. $\gamma = 5$), this overhead diminishes and becomes a ratio of ≈ 0.55 (resp. ≈ 0.21) the size of the input key file.

The first strategy is the default strategy proposed in our implementation. The second one has also been implemented and can be optionally switched on.

3.5 Termination

The expected number of unplaced keys decreases exponentially with the number of levels but is not theoretically guaranteed to reach zero in a finite number of steps. To ensure termination of the construction algorithm, in our implementation a maximal number D of levels is fixed. Then, the remaining keys are inserted into a regular hash table. Value D is a parameter, its default value is $D = 25$ for which the expected number of keys stored in this hash table is $\approx 10^{-5}N$ for $\gamma = 1$ and becomes in practice negligible for $\gamma \geq 2$, allowing the size overhead of the final hash table to be negligible regarding the final `MPHF` size.

4 Results

We evaluated the performance *BBhash* for the construction of large `MPHF`s. We generated files containing various numbers of keys (from 1 million to 1 trillion keys). In our tests, a key is a binary representation of a pseudo-random positive integer in $[0; 2^{64}]$. Within each file, each key is unique. We also performed a test where input keys are strings (n-grams) to ensure that using integers as keys does not bias results. Tests were performed on a cluster node with a Intel© Xeon© CPU E5-2660 v3 2.60GH 20-core CPU, 256 GB of memory, and a mechanical hard drive. Except for the experiment with 10^{12} keys, running times include the time needed to read input keys from disk. Note that files containing key sets may be cached in memory by the operating system, and all evaluated methods benefit from this effect during `MPHF` construction. We refer to the Appendix for the specific commands and parameters used in these experiments.

We first analyzed the influence of the γ value (the main parameter of *BBhash*), then the effect of using multiple threads depending on the parallelization strategy. Second, we compared *BBhash* with other state-of-the-art methods. Finally, we performed an `MPHF` construction on 10^{12} elements.

4.1 Influence of the γ parameter

We report in Figure 2 (left) the construction times and the mean query times, as well as the size of the produced `MPHF`, with respect to several γ values. The main observation is that $\gamma \geq 2$ drastically accelerates construction and query times. This is expected since large γ values allow more elements to be placed in the first levels of the `MPHF`; thus limiting the number of times each key is hashed to compute its level. In particular, for keys placed in the very first level, the query time is limited to a single hashing and a memory access. The average level of all keys is $e^{(1/\gamma)}$, we therefore expect construction and query times to decrease when γ increases. However, larger γ values also incur larger `MPHF` sizes. One observes that $\gamma > 5$ values seem to bring very little advantage at the price of higher space requirements. A related work used $\gamma = 1$ in order to minimize the `MPHF` size [6]. Here, we

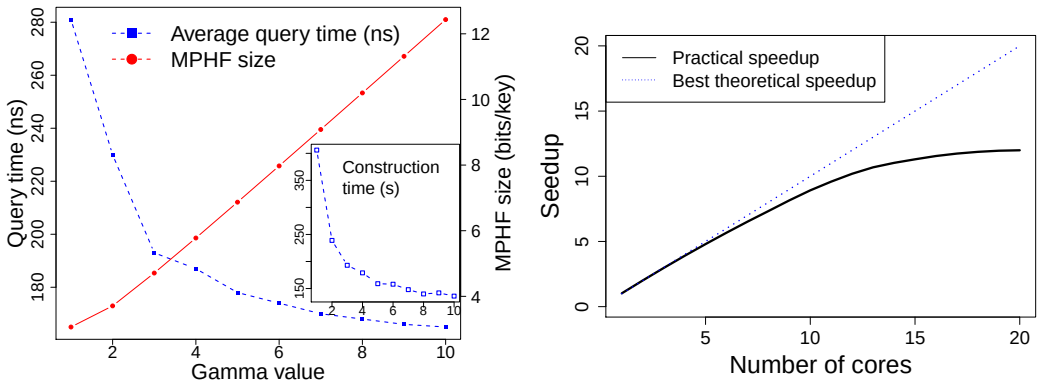

Figure 2 Left: Effects of the gamma parameter on the performance of *BBhash* when run on a set composed of one billion keys, when executed on a single CPU thread. Times and `MPHF` size behave accordingly to the theoretical analysis, respectively $O(e^{(1/\gamma)})$, and $O(\gamma e^{(1/\gamma)})$. Right: Performance of the *BBhash* construction time according to the number of cores, using $\gamma = 2$.

argue that using γ values larger than 1 has significant practical merits. In our tests, we often used $\gamma = 2$ as it yields an attractive time/space trade-off during construction and queries.

4.2 Parallelization performance

We evaluated the capability of our implementation to make use of multiple CPU cores. In Figure 2 (right), we report the construction times with respect to the number of threads. We observe a near-ideal speed-up with respect to the number of threads with diminishing returns when using more than 10 threads, which is likely due to cache misses that induce a memory access bottleneck.

In addition to these results, we applied *BBhash* on a key set of 10 billion keys and on a key set of 100 billion keys, again using default parameters and 8 threads. The memory usage was respectively 4.96GB and 49.49GB, and the construction time was respectively 462 seconds and 8913 seconds, showing the scalability of *BBhash*.

4.3 Comparisons with state of the art methods

We compared *BBhash* with state-of-the-art `MPHF` methods. CHD (http://cmph.sourceforge.net/) is an implementations of the compressed hash-and-displace algorithm [2]. EMPHF [1] is based on random hypergraph peeling, and the HEM [4] implementation in EMPHF is based on partitioning the input data. Sux4J is a Java implementation of [11]. We did not include other methods cited earlier because they do not provide an implementation [12, 16] or the software integrates a non-minimal perfect hash function that is not stand-alone [6]. However single-threaded results presented in [16] show that construction times and `MPHF` sizes are comparable to ours, query times are significantly longer, and no indication is provided about the memory usage during construction. Our benchmark code is available at https://github.com/rchikhi/benchmphf.

Figure 3 shows that all evaluated methods are able to construct `MPHF`s that contain a billion elements, but only *BBhash* scales up to datasets that contain 10^{11} elements and more. Overall, *BBhash* shows consistently better time and memory usage during construction.

We additionally compared the resulting `MPHF` size, i.e. the space of the data structure

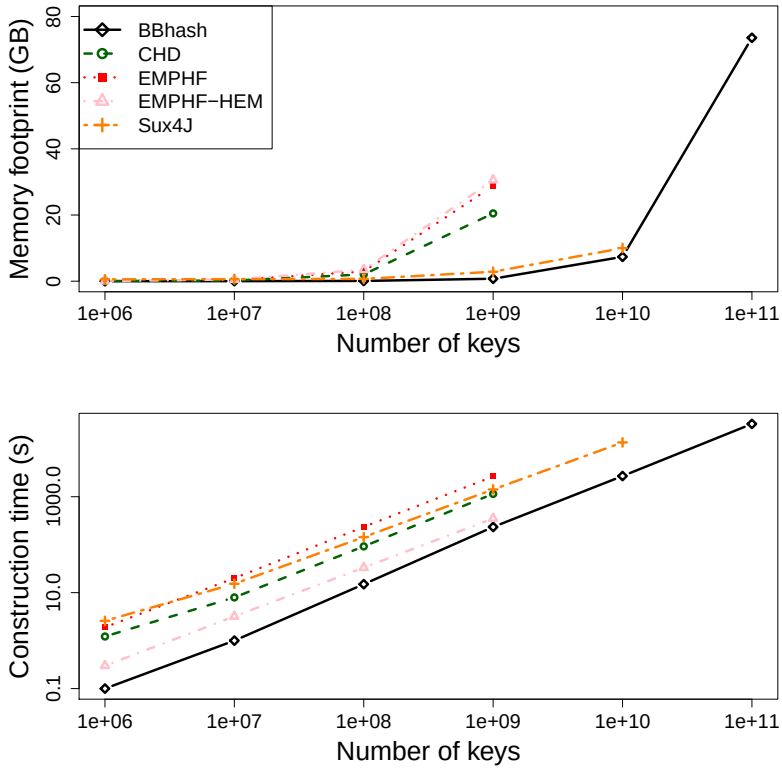

Figure 3 Memory footprint and construction time with respect to the number of keys. All libraries were run using default parameters, including $\gamma = 2$ for *BBhash*. For a fair comparison, *BBhash* was executed on a single CPU thread. Except for Sux4J, missing data points correspond to runs that exceeded the amount of available RAM. Sux4J limit comes from the disk usage, estimated at approximately 4TB for 10^{11} keys.

returned by the construction algorithm, and the mean query time across all libraries on a dataset consisting of a billion keys (Table 1). `MPHF`s produced by *BBhash* range from 2.89 bits/key (when $\gamma = 1$ and ranks are sampled every 1024 positions) to 6.9 bits/key (when $\gamma = 5$ and a rank sampling of 512). The 0-0.8 bits/key size difference between our implementation and the theoretical space usage of the *BBhash* structure size is due to additional space used by the rank structure. We believe that a reasonable compromise in terms of query time and structure size is 3.7 bits/key with $\gamma = 2$ and a rank sampling of 512, which is marginally larger than the `MPHF` sizes of other libraries (ranging from 2.6 to 3.5 bits/key). As we argued in the Introduction, using one more bit per key seems to be a reasonable trade-off for performance.

Construction times vary by one or two orders of magnitude across methods, *BBhash* being the fastest. With default parameters ($\gamma = 2$, rank sampling of 512), *BBhash* has a construction memory footprint 40× to 60× smaller than other libraries except for Sux4j, for which *BBhash* remains 4× smaller. Query times are roughly within an order of magnitude (179 − 1037 ns) of each other across methods, with a slight advantage for *BBhash* when $\gamma \geq 2$. Sux4j achieves an attractive balance with low construction memory and query times, but high disk usage. In our tests, the high disk usage of Sux4j was a limiting factor for the construction of very large `MPHF`s.

Method	Query time (ns)	MPHF size (bits/key)	Const. time* (s)	Const. memory**	Disk. usage (GB)
BBhash $\gamma = 1$	271	3.1	60 (393)	3.2 (376)	8.23
BBhash $\gamma = 1$ minirank	279	2.9	61(401)	3.2 (376)	8.23
BBhash $\gamma = 2$	216	3.7	35 (229)	4.3 (516)	4.45
BBhash $\gamma = 2$ nodisk	216	3.7	80 (549)	6.2 (743)	0
BBhash $\gamma = 5$	179	6.9	25 (162)	10.7 (1,276)	1.52
EMPHF	246	2.9	2,642	247.1 (29,461)†	20.8
EMPHF HEM	581	3.5	489	258.4 (30,798)†	22.5
CHD	1037	2.6	1,146	176.0 (20,982)	0
Sux4J	252	3.3	1,418	18.10 (2,158)	40.1

Table 1 Performance of different MPHF algorithms applied on a key set composed of 10^9 64-bits random integers, of size 8GB. Each time result is the average value over three tests. The 'nodisk' row implements the second strategy described in Section 3.4, and the 'minirank' row samples ranks every 1024 positions instead of 512 by default. *The column "*Const. time*" indicates the construction time in seconds. In the case of *BBhash*, the first value is the construction time using eight CPU threads and the second value in parenthesis is the one using one CPU thread. **The column "*Const. memory*" indicates the RAM used during the MPHF construction, in bits/key and the total in MB in parenthesis. † The memory usages of EMPHF and EMPHF HEM reflect the use of memory-mapped files (mmap scheme).

Note that EMPHF, EMPHF HEM and Sux4j implement a disk partitioning strategy, that could in principle also be applied to others methods, including ours. Instead of creating a single large MPHF, they partition the set of input keys on disk and construct many small MPHFs independently. In theory this technique allows to engineer the MPHF construction algorithm to use parallelism and lower memory, at the expense of higher disk usage. In practice we observe that the existing implementations that use this technique are not parallelized. While EMPHF and EMPHF HEM used relatively high memory in our tests (around 30 GB for 1 billion elements) due to memory-mapped files, they also completed the construction successfully on another machine that had 16 GB of available memory. However, we observed what appears to be limitations in the scalability of the scheme: we were unable to run EMPHF and EMPHF HEM on an input of 10 billion elements using 256 GB of memory. Regardless, we view this partitioning technique as promising but orthogonal to the design of efficient "monolithic" MPHFs constructions such as *BBhash*.

4.4 Performance on an actual dataset

In order to ensure that using pseudo-random integers as keys does not bias results, we ran *BBhash* using strings as keys. We used n-grams extracted from the Google Books Ngram dataset[1], version 20120701. On average the n-gram size is 18. We also generated random words of size 18. As reported in Table 2, we obtained highly similar results to those obtained with random integer keys.

[1] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Dataset	Query time (ns)	MPHF size (bits/key)	Const. time (s)
10^8 Random strings	325	3.7	35
10^8 Ngrams	296	3.7	37

Table 2 Performance of *BBhash* ($\gamma = 2$, 8 threads) when using ASCII strings as keys.

4.5 Indexing a trillion keys

We performed a very large-scale test by creating an MPHF for 10^{12} keys. For this experiment, we used a machine with 750 GB of RAM. Since storing that many keys would require 8 TB of disk space, we instead used a procedure that deterministically generates a stream of 10^{12} pseudo-random integers in $[0, 2^{64} - 1]$. We considered the streamed values as input keys without writing them to disk. In addition, key sets of cardinality below 20 billion (2% of the input) were stored in memory to avoid re-computation from scratch at each subsequent level. Thus, the reported computation time should not be compared to previously presented results as this experiment has no disk accesses. The test was performed using $\gamma = 2$, 24 threads.

Creating the MPHF took 35.4 hours and required 637 GB RAM. This memory footprint is roughly separated between the bit arrays ($\approx$ 459 GB) and the memory required for loading 20 billion keys in memory ($\approx$ 178 GB). The final MPHF occupied 3.71 bits per key.

5 Conclusion

We have proposed a resource-efficient and highly scalable algorithm for constructing and querying MPHFs. Our algorithmic choices were motivated by simplicity: the method only relies on bit arrays and classical hash functions. While the idea of recording collisions in bit arrays to create MPHFs is not novel [6, 12], to the best of our knowledge *BBhash* is the first implementation that is competitive with the state of the art. The construction is particularly time-efficient as it is parallelized and mainly consists in hashing keys and performing memory accesses. Moreover, the additional data structures used during construction are provably small enough to ensure a low memory overhead during construction. In other words, creating the MPHF does not require much more space than the resulting MPHF itself. This aspect is important when constructing MPHFs on large key sets in practice.

Experimental results show that *BBhash* generates MPHFs that are slightly larger to those produced by other methods. However *BBhash* is by far the most efficient in terms of construction time, query time, memory and disk footprint for indexing large key sets (of cardinality above 10^9 keys). The scalability of our approach was confirmed by constructing MPHFs for sets as large as 10^{12} keys. To the best of our knowledge, no other MPHF implementation has been tested on that many keys.

A time/space trade-off is achieved through the γ parameter. The value $\gamma = 1$ yields MPHFs that occupy roughly $3N$ bits of space and have little memory overhead during construction. Higher γ values use more space for the construction and the final structure size, but they achieve faster construction and query times. Our results suggest that $\gamma = 2$ is a good time-versus-space compromise, using 3.7 bits per key. With respect to hypergraph-based methods [1, 3, 4, 11], *BBhash* offers significantly better construction performance, but the resulting MPHF size is up to 1 bit/key larger. We however argue that the MPHF size, as long as it is limited to a few bits per key, is generally not a bottleneck as many applications use MPHFs to associate much larger values to keys. Thus, we believe that this work will unlock many high performance computing applications where the possibility to index billions keys

and more is a huge step forward.

An interesting direction for future work is to obtain more space-efficient MPHFs using our method. We believe that a way to achieve this goal is to slightly change the hashing scheme. We would like to explore an idea inspired by the CHD algorithm for testing several hash functions at each level and selecting (then storing) one that minimizes the number of collisions. At the price of longer construction times, we anticipate that this approach could significantly decrease the final structure size.

Acknowledgments

This work was funded by French ANR-12-BS02-0008 Colib'read project. We thank the GenOuest BioInformatics Platform that provided the computing resources necessary for benchmarking. We thank Djamal Belazzougui for helpful discussions and pointers.

References

1 Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Data Compression Conference (DCC), 2014*, pages 352–361. IEEE, 2014.

2 Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

3 Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Algorithms and Data Structures*, pages 139–150. Springer, 2007.

4 Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.

5 Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005. doi:10.1093/comjnl/bxh074.

6 Jarrod A Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P Schroth, and Daniel S Rokhsar. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011.

7 Yupeng Chen, Bertil Schmidt, and Douglas L Maskell. A hybrid short read mapping accelerator. *BMC Bioinformatics*, 14(1):67, 2013. doi:10.1186/1471-2105-14-67.

8 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

9 Zbigniew J Czech, George Havas, and Bohdan S Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1):1–143, 1997.

10 Michael L Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

11 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In V. Andrew Goldberg and S. Alexander Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 339–352. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-38851-9_23.

12 Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.

13 George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

14 Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 170–175. IEEE, 1982.

15 Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 746–755. Society for Industrial and Applied Mathematics, 2008.

16 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. *Retrieval and Perfect Hashing Using Fingerprinting*, pages 138–149. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-07959-2_12.

Appendix

Proofs of MPHF size and memory required for construction

MPHF size with $\gamma = 1$.

$$\begin{aligned}\sum_{d\geq 0}|A_d| &= N\sum_{d\geq 0}(1-e^{-1})^d \\ &= N\frac{1}{1-(1-e^{-1})} && \text{as } \lim_{d\to+\infty}(1-e^{-1})^d = 0 \\ &= eN\end{aligned}$$

◀

MPHF size using any $\gamma \geq 1$. With $\gamma \geq 1 : |A_d| = \gamma|A_{d-1}|(1-e^{\frac{-1}{\gamma}}) = \gamma|A_0|(1-e^{\frac{-1}{\gamma}})^d = \gamma N(1-e^{\frac{-1}{\gamma}})^d$

$$\text{Thus, } \sum_{d\geq 0}|A_d| = \gamma N\sum_{d\geq 0}(1-e^{\frac{-1}{\gamma}})^d$$

Moreover, as $\lim_{d\to+\infty}(1-e^{\frac{-1}{\gamma}})^d = 0$ since for $\gamma > 0, 0 < 1-e^{\frac{-1}{\gamma}} < 1$, on has:

$$\sum_{d\geq 0}|A_d| = \gamma N\frac{1}{1-(1-e^{\frac{-1}{\gamma}})} = \gamma e^{\frac{1}{\gamma}}N$$

◀

Note that this proof stands for any γ value > 0, but that with $\gamma < 1$ the theoretical and practical MPHF sizes increase exponentially as γ get close to zero.

Lemma 1. Let $m(d)$ be memory required during level d and let R be the ratio between the maximal memory needed during the MPHF construction and the MPHF total size denoted by S. Formally,

$$R = \frac{\max_{d\geq 0}(m(d))}{S} = \frac{\max_{d\geq 0}(m(d))}{\gamma e^{\frac{1}{\gamma}}N}$$

First we prove that $\lim_{d\to\infty}\frac{m(d)}{S} = 1$.

$$m(d) = \sum_{i<d}|A_i| + 2|A_d| = \gamma N\left(\frac{1-(1-e^{\frac{-1}{\gamma}})^d}{e^{\frac{-1}{\gamma}}} + 2(1-e^{\frac{-1}{\gamma}})^d\right)$$

Since for $\gamma > 0$, $0 < 1-e^{\frac{-1}{\gamma}} < 1$, then $\lim_{d\to\infty} m(d) = \gamma e^{\frac{1}{\gamma}}N$. Thus $\lim_{d\to\infty}\frac{m(d)}{S} = 1$. Before going further, we need to compute $m(d+1) - m(d)$:

$$\begin{aligned}m(d+1) - m(d) &= \sum_{i<d+1}|A_i| + 2|A_{d+1}| - \sum_{i<d}|A_i| + 2|A_d| \\ &= |A_d| + 2|A_{d+1}| - 2|A_d| = 2|A_{d+1}| - |A_d| \\ &= 2\gamma N(1-e^{\frac{-1}{\gamma}})^{d+1} - \gamma N(1-e^{\frac{-1}{\gamma}})^d \\ &= \gamma N(1-e^{\frac{-1}{\gamma}})^d(2(1-e^{\frac{-1}{\gamma}}) - 1) \\ &= \gamma N(1-e^{\frac{-1}{\gamma}})^d(1-2e^{\frac{-1}{\gamma}})\end{aligned}$$

We now prove $R \leq 1$ when $\gamma \leq \frac{1}{\log(2)}$ and also, $R < 2$ when $\gamma > \frac{1}{\log(2)}$.

- Case 1: $\gamma \leq \frac{1}{\log(2)}$
  We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}} \leq 2e^{-\log(2)} = 1$.
  Moreover, as $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d(1 - 2e^{\frac{-1}{\gamma}})$ and as, with $\gamma \leq \frac{1}{\log(2)}$: $1 - e^{\frac{-1}{\gamma}} \geq 0.5$, and $1 - 2e^{\frac{-1}{\gamma}} \geq 0$ then $m(d+1) - m(d) \geq 0$, thus, m is an increasing function.
  To sum up, with $\gamma \leq \frac{1}{\log(2)}$, we have **1/** that $\frac{m(0)}{S} \leq 1$, **2/** that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$, and **3/** that m is increasing, then $R \leq 1$.
- Case 2: $\gamma > \frac{1}{\log(2)}$ We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}}$. With $\gamma > \frac{1}{\log(2)}$, $1 < \frac{m(0)}{S} < 2$. Moreover, $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d(1 - 2e^{\frac{-1}{\gamma}})$ is negative as: $1 - e^{\frac{-1}{\gamma}} > 0$ and $1 - 2e^{\frac{-1}{\gamma}} < 0$ for $\gamma > \frac{1}{\log(2)}$. Thus m is a decreasing function with d.
  With $\gamma > \frac{1}{\log(2)}$, we have **1/** that $\frac{m(0)}{S} < 2$, **/2** that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$ and **/3** that m is decreasing. Thus $R < 2$.

◀

Algorithms pseudo-codes

Algorithm 1: `MPHF` construction.

Data: F_0 a set of N keys, integers γ and $last$
Result: array of bit arrays $\{A_0, A_1, \ldots, A_{last}\}$, hash table H
i=0;
while F_i *not empty and* $i \leq last$ **do**
  $A_i = ArrayFill(F_i, \gamma)$;
  foreach *key* x *of* F_i **do**
    $h = hash(x) \mod (\gamma * N)$;
    if $A_i[h] == 0$ **then**
      $F_{i+1}.add(x)$
  i=i+1;
Construct H using remaining elements from F_{last+1};
Return $\{A_0, A_1, \ldots, A_{last}, H\}$

In practice F_i with $i > 1$ are stored on disk (see Section 3.4). The hash table H ensures that elements in F_{last+1} are mapped without collisions to integers in $[|F_0| - |F_{last+1}| + 1, |F_0|]$

Algorithm 2: *ArrayFill*

Data: F array of N keys, integer γ
Result: bit array A
Zero-initialize A and C two bit arrays with $\gamma * N$ elements;
foreach *key* x *of* F **do**
  $h = hash(x) \mod (\gamma * N)$;
  if $A[h] == 0$ *and* $C[h] == 0$ **then**
    $A[h] = 1$;
  if $A[h] == 1$ *and* $C[h] == 0$ **then**
    $A[h] = 0$;
    $C[h] = 1$;
  if $A[h] == 0$ *and* $C[h] == 1$ **then**
    Skip;
Delete C;
Return A;

Note that the case $A[h] == 1$ and $C[h] == 1$ never happens.

Algorithm 3: `MPHF` query

Data: bit arrays $\{A_0, A_1, \ldots, A_{\text{last}}\}$, hash table H, key x
Result: integer index of x
i=0;
while $i \leq last$ **do**
  $h = hash_i(x) \mod A_i.size()$;
  if $A_i[h] == 1$ **then**
    return $\sum_{j<i} |A_j| + rank(A_i[h])$;
  $i = i + 1$;
return *H[x]* ;

Note, when x is not an element from the key set of the `MPHF`, the algorithm may return a wrong integer index.

Commands

In this section we describe used commands for each presented result. Time and memory usages where computed using "*/usr/bin/time –verbatim*" unix command. The disk usage was computed thanks to a home made script measuring each 1/10 second the size of the directory using the "*du -sk*" unix command, and recording the highest value. The *BBhash* library and its *Bootest* tool are available from https://github.com/rizkg/BBHash.

Commands used for Section 4.1:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

Note that 1000000000 is the number of keys tested and 1 is the number of used cores.

Additional tests, with larger key set and 8 threads:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

Commands used for Section 4.2:

```
for keys in 10000000000 100000000000; do
./Bootest ${keys} 8 2 -bench
done
```

Commands used for Section 4.3:

We remind that our benchmark code, testing EMPHF, EMPHF MEM, CHD, and Sux4J is available at https://github.com/rchikhi/benchmphf.

- *BBhash* commands:
  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./Bootest ${keys} 1 2 -bench
  done
  ```
- *BBhash* command with nodisk (Table 1) was
  `./Bootest 1000000000 1 2 -bench -nodisk`
  and
  `./Bootest 1000000000 8 2 -bench -nodisk`
  respectively for one and height threads. Other commands from Table 1 were deduced from previously presented *BBhash* computations.
- Commands EMPHF & EMPHF HEM:
  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./benchmphf ${keys} -emphf
  done
  ```
  EMPHF (resp. EMPHF HEM) is tested by using the #define EMPHF_SCAN macro (resp. #define EMPHF_HEM). In order to assess the disk size footprint, the line "*unlink(tmpl);*" from file "*emphf/mmap_memory_model.hpp*" was commented.
- Commands CHD:

```
for keys in 1000000 10000000 100000000 10000000000\
 10000000000 100000000000; do
./benchmphf ${keys} -chd
done
```

- Commands Sux4J:
  for each size, the *"Sux4J/slow/it/unimi/dsi/sux4j/mph/LargeLongCollection.java"* was modified indicating the used size.
  `./run-sux4j-mphf.sh`

Commands used for Section 4.4:

As explained Section 4.4, the keyString.txt file is composed of n-grams extracted from the Google Books Ngram dataset[2], version 20120701.
`./BootestFile keyStrings.txt 10 2`

Commands used for Section 4.5:

BBhash command for indexing a trillion keys, with keys generated on the fly.
`./Bootest 1000000000000 24 2 -onthefly`

[2] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html