Brief Announcement: Using Multi-Level Parallelism and 2-3 Cuckoo Filters for Set Intersection Queries and Sparse Boolean Matrix Multiplication

David Eppstein
Univ. of California, Irvine
eppstein@uci.edu

Michael T. Goodrich
Univ. of California, Irvine
goodrich@uci.edu

ABSTRACT

We use multi-level parallelism and a new type of data structures, known as ***2-3 cuckoo*** filters, to answer set intersection queries faster than previous methods, with applications to improved sparse Boolean matrix multiplication.

KEYWORDS

set intersection, 2-3 cuckoo filters, Boolean matrix multiplication

1 INTRODUCTION

Assume we have a collection, $S_1, S_2, \ldots$, of sets, of variable sizes, which can be preprocessed and represented in some canonical data structures stored in main memory. In a ***set-intersection query***, we are then given the names of two sets, S_i and S_j, and are asked to produce a listing of the members in the common intersection, $S_i \cap S_j$. As discussed by Amossen and Pagh [2], such queries have applications in sparse Boolean matrix multiplication.

In this paper, we provide asymptotic improvements to such set-intersection queries, for pairs of sets of different sizes, by taking advantage of bit-level parallelism, i.e., an ability to compute AC^0 functions on pairs of binary words in constant time, including such operations as AND, OR, XOR, and MSB (most-significant set bit). This computational model is known as the ***practical RAM*** model [11], and we refer to a parallel shared-memory extension of this model as the ***practical PRAM*** model.

Related Work. In 1996, Miltersen [11] introduced the ***practical RAM*** model, which has inspired futher research for algorithms in this model (e.g., see [3, 12]). Bille *et al.* [4] present a data structure that can compute the intersection of t sets of total size n in expected time $O(n(\log^2 w)/w + kt + \log w)$. Kopelowitz *et al.* [9] introduce a data structure for computing set intersections for pairs of sets of roughly the same size and use it to list the triangles in a graph G in $O(m\lceil(\alpha(G)\log^2 w)/w + \log w\rceil + k)$ expected time, where $\alpha(G)$ denotes the arboricity of G. Eppstein *et al.* [6] improve this bound to $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$ expected time, using a data structure they call "2-3 cuckoo hash-filters," but their construction is likewise limited to pairwise intersections of sets of roughly the same size.

SPAA '17, July 2017, Washington, DC USA

ACM ISBN 978-1-4503-4593-4/17/07.
https://doi.org/10.1145/3087556.3087599

Lingas [10] shows how to multiply two $n \times n$ Boolean matrices in time $O(n^2 s^{0.188})$, where s is the number of non-zero entries in the output matrix. Amossen and Pagh [1, 2] also study the sparse Boolean matrix multiplication problem, and approach it, as we do, as an application of set-intersection data structures. They show how to perform Boolean matrix multiplication in time $O(m^{0.86}s^{0.41} + (ms)^{2/3})$, where m is the number of non-zero entries in the input matrices [1].

Our Results. We show how to use 2-3 cuckoo hash tables and filters to quickly compute the common intersection of pairs of sets of varying sizes, with applications to sparse Boolean matrix multiplication. We show that these two simple ideas allow us to compute set-intersection queries for pairs of sets of total size n in expected time $O(n(\log w)/w + k)$. This, in turn, leads to new algorithms for sparse Boolean matrix multiplication running in expected time $O(n^2 + nm(\log w)/w)$. Finally, we show how to implement our solutions to compute set-intersection queries in $O(\log n)$ time in the practical PRAM model.

2 2-3 CUCKOO HASH-FILTERS

We begin by reviewing the 2-3 cuckoo hash-filter data structure of Eppstein *et al.* [6], Suppose, then, we wish to maintain a set, S, of n elements taken from a universe such that each element can be stored in a single memory word, where $w \geq \log n$. We maintain S using the following three components, T, C, and F:

- A hash table T of size $O(n)$, using three pseudo-random hash functions h_1, h_2, and h_3, which map elements of S to triples of distinct integers in the range $[0, n-1]$. Each element x in S is stored, if possible, in two of the three possible locations for x based on these hash functions.
- We also store a stash cache [8], C, of size λ, where λ is bounded by a constant. C stores elements for which it was not possible to store properly in two distinct locations in T.
- A table, F, having $O(n)$ cells, that parallels T, so that $F[j]$ stores a non-zero fingerprint digest, $f(x)$, for an element x, if and only if $T[j]$ stores a copy of x. The digest $f(x)$ is a non-zero random hash function comprising δ bits, where $\delta = \Theta(\log w)$. The table F is called a 2-3 ***cuckoo filter***, and it is stored in a packed format, so that we store $O(w/\log w)$ cells of F per memory word. In addition to the vector F, we store a bit-mask, M, that is the same size as F and has all 1 bits in the corresponding cell of F that is occupied.

We assume we can read and write individual cells of F and M in $O(1)$ time. These cells amount to subfields of words of $O(\log w)$ size, which can be read from or written to using standard bit-level operations, such as AND, OR, XOR, etc.

Parallel Construction Algorithm. Since the method for constructing a 2-3 cuckoo filter, F, is the same as that for a 2-3 hash table, T, that is parallel to it, without loss of generality, let us describe how to construct T. We assume we have n elements that need to be added to T and that T has size that is at least $6(1+\epsilon)n$, for a constant $\epsilon > 0$. We also assume that we have a stash cache, C, of constant size, λ. Eppstein *et al.* [6] describe a sequential algorithm for constructing a 2-3 cuckoo hash-filter.

We describe here a parallel algorithm, which can easily be simulated sequentially if one desires a sequential algorithm (which would only use bit-level parallelism). Different than Eppstein *et al.*, we proceed in a sequence of rounds. In each round, we have some set, S, of elements that have yet to be added to T. Initially, S is the entire set of n elements, where we create two copies of each element, where each copy functions somewhat independently. In a given round, each element in S has 3 locations in T where it can be placed, with at most one of these places being a location where it was previously displaced (initially there are no such previous locations). For each such element, x, we read each of the 2 (or initially 3) locations, $T[h_i(x)]$, where it was not previously displaced, and consider the following possible cases: (1) If one of these locations, $T[h_i(x)]$ is empty, then we choose one of these empty locations at random and try to inject x at that location. (2) Otherwise, we choose a location, $T[h_i(x)]$, at random, that does not already hold x and we try to inject x into that location.

Since we are implementing this computation in parallel in the practical PRAM model, we need to deal with the likely possibility that the injection step performed for multiple elements simultaneously might involve concurrent writes; hence, we assume we are operating in the practical CRCW PRAM model where write conflicts are resolved arbitrarily. After we perform this injection step in $O(1)$ time, we collect all the elements that failed to be injected, and let this set be the set S for the next round. This collection step can easily be done in $O(\log n)$ time using $O(n)$ work (e.g., see [7]). We repeat this process for $D \log n$ rounds, where D is a constant. If the size of the set S at the end of this process is at most λ, then we set the stash cache equal to S (which could even be empty) and we consider the construction of T a success. Otherwise, we consider the construction of T to be a failure.

If the construction of T succeeds, then we create a parallel (i.e., mirrored) copy of T as a hash filter, F, by replacing each element with its fingerprint of size $O(\log w)$ and compressing every block of size $O(w/\log w)$ words in T into a single work for F. If the construction of T fails, however, then we instead sort the elements of the original set, S, of n elements and just use this sorted copy of S to represent S. Following the analysis of Eppstein *et al.* [6], we can show:

Theorem 2.1. *For any constant integer $s \geq 1$, the (constant) size, s, S of the stash in a 2-3 cuckoo hash table after all items have been inserted satisfies $\Pr(S \geq s) = \tilde{O}(n^{-s})$, where $\tilde{O}$ ignores polylogarithmic factors.*

3 LISTS OF 2-3 CUCKOO HASH-FILTERS

We construct our data structure for S as follows.

1. Sort the elements of S according to a global total order for all sets, and divide this sorted listing of S into intervals, $I_1, I_2, \ldots,$ such that each interval I_j contains a subset, S_j, of S of $O(w/\log w)$ elements.

2. For each subset S_j in parallel, use the algorithm given above to construct a 2-3 cuckoo hash-filter for S_j with a stash of constant size, λ. If the construction for some S_j fails, then simply fallback to representing this subset as a sorted listing of its elements.

3. For each subset, S_j, that had a failed cuckoo construction, subdivide S_j into smaller intervals, $I_{j,1}, I_{j,2}, \ldots$, such that each interval contains exactly one element of S_j, and construct a single-element 2-3 cuckoo hash-filter for each.

4. For each subset, S_j, that had a successful cuckoo construction, subdivide S_j into $O(\lambda)$ smaller intervals, $I_{j,1}, I_{j,2}, \ldots$, so that each interval contains at most one stash element. Then copy the 2-3 cuckoo filter for S_j into $O(\lambda)$ copies, one for each $I_{j,k}$ interval, removing the elements of S_j that do not belong to $I_{j,k}$, so that no 2-3 cuckoo filter in this group has a non-empty stash.

Given our construction methods, and known results for parallel sorting (e.g., see [7]), it is straightforward to show that we can construct the above representation for each set in a collection of total size n in $O(\log n)$ time using expected work $O(n \log n)$ in the practical (CRCW) PRAM model.

Intersecting 2-3 Cuckoo Hash-Filters. Let us review the method of Eppstein *et al.* [6] for intersecting a pair of 2-3 cuckoo hash-filters that are the same size. Suppose then that we have two subsets, S_i, and S_j, for which we wish to compute a representation of the intersection of these two sets. We begin our set-intersection algorithm by computing a vector of $O(1)$ words that identifies the matching non-empty cells in F_i and F_j. For example, we could compute the vector defined by the following bit-wise vector expression:

$$A = (M_i \text{ AND NOT } (F_i \text{ XOR } F_j)). \quad (1)$$

We view A as being a parallel vector to F_i and F_j. Note that a cell, $A[r]$, consists of δ bits and this cell is all 1s if and only if $F_i[r]$ stores a fingerprint digest for some element and $F_i[r] = F_j[r]$, since fingerprint digests are non-zero. Thus, we can create the list, L, of members of the common intersection of S_i and S_j, by visiting each word of A and storing to L the element in $T_i[r]$ corresponding to each cell, $A[r]$, that is all 1s, but doing so only after confirming that $T_i[r] = T_j[r]$. Doing the listing can be done in time $O(1+k+t)$, where k is the number of elements in the intersection and t is the number of false positives, by using bit-level operations in the practical RAM model (e.g., see [5]).

Answering Set-Intersection Queries. Let us next describe our algorithm for answering a query asking for the intersection of two sets, S_1 and S_2, of possibly different sizes. Let n_1 (n_2)denote the size of S_1 (S_2), and let $n = n_1 + n_2$.

Let us assume that S_1 and S_2 are each represented using the lists of 2-3 cuckoo hash-filters. Furthermore, as explained above, there are no stashes and every subset has a hash-filter. The goal of our algorithm is to compute a cuckoo hash-filter representation that contains all the elements in $S_1 \cap S_2$, plus possibly some false positives that our algorithm identifies as high-probability elements belonging to this common intersection. After we have performed the core part of algorithm, then, we can simply go through the list of cuckoo-filters and confirm which elements actually belong to the common intersection.

Because of the way our algorithm works, it produces a cuckoo-filter representation for the common intersection, where $S_1 \cap S_2$ is represented as a sorted list of intervals (no two consecutive of which are empty), such that for each interval, I, we have a cuckoo filter, F_I, and a backing 2-3 cuckoo table, T_I, where, for each non-zero fingerprint, $F_I[j]$, there is a corresponding element, $x = T_I[j]$, with x being a confirmed element in S_1. The cuckoo filter, F_I, might not be a 2-3 cuckoo filter, however. Nevertheless, after our core algorithm completes, our representation allows us to examine each non-zero cuckoo filter fingerprint in a filter F_I, lookup its corresponding element, x, in a backing 2-3 cuckoo table, T_I, and then perform a search for x in the global hash table, H_2 for S_2. Thus, after one additional lookup for each such element, x, we can confirm or discard x depending on whether it is or isn't in the common intersection.

Our algorithm for constructing this representation of $S_1 \cap S_2$, and then culling out false positives, is as follows.

1. Merge the list of interval boundaries for S_1 and S_2 according to the global total order for sets. This step can be done in parallel in $O(\log n)$ time and $O(n(\log w)/w)$ work.

2. For each overlapping interval, $I_{1,j}$ and $I_{2,k}$, where $I_{1,j}$ is from S_1 and $I_{2,k}$ is from S_2, intersect these two subsets using the bit-parallel intersection algorithm for 2-3 cuckoo filters derived using Equation 1 above (but skipping the lookups in the corresponding 2-3 cuckoo table and any lookups for elements in stashes). Let $F_{j,k}$ denote the resulting (now partial) 2-3 cuckoo filter. This step can be implemented in parallel in $O(1)$ time and $O(n(\log w)/w)$ work, since the total number of overlapping pairs of intervals is $O(n(\log w)/w)$.

3. For each interval, $I_{1,j}$, collect all the partial 2-3 cuckoo hash-filters, $F_{j,k}$, computed in the previous step for $I_{1,j}$. Compute the bit-wise OR of these filters. Let F_j denote the resulting partial 2-3 cuckoo filter for $I_{1,j}$, and let $\mathbf{F}$ denote the collection of all such filters (note that there is potentially a non-empty partial cuckoo filter, F_j, for each interval in S_1). This step can be implemented in parallel in $O(1)$ time and $O(n(\log w)/w)$ work, since the total number of overlapping pairs of intervals is $O(n(\log w)/w)$.

4. For each interval $I_{1,j}$, and each element, x, in the 2-3 cuckoo hash table for $I_{1,j}$ that has a corresponding fingerprint belonging to F_j in $\mathbf{F}$, do a lookup in each of the global hash tables H_1 and H_2 to determine if x is indeed a common element in the sets S_1 and S_2. Let Z denote the set of all such elements so determined to belong to this common intersection. This step can be implemented in $O(\log n)$ time and $O(n(\log w)/w + k + p)$ work, where k is the size of the output and p is the number of false positives, that is, elements that have a non-zero fingerprint in some F_j but nevertheless are not in the common intersection, $S_1 \cap S_2$.

Theorem 3.1. *Given two sets, S_1 and S_2, represented using ultra-compact 2-3 cuckoo hash-filters, one can compute a listing of the k elements in $S_1 \cap S_2$ in $O(\log n)$ time using $O(n(\log w)/w + k)$ expected work in the practical CRCW PRAM model (or sequentially in this expected time).*

Sparse Boolean Matrix Multiplication. Suppose we are given two Boolean $n \times n$ matrices, A and B, and asked to compute $C = A \times B$, where the scalar plus (+) operation is Boolean OR and the scalar times operation is Boolean AND. Our method for producing C involves using the above data structure to represent the indices of the non-zero elements in each row of A and each column of B. Producing these representations using ultra-compact 2-3 cuckoo hash-filters can be done in $O(\log n)$ time and $O(n)$ expected work in the practical CRCW PRAM model, for each such row and column, because we don't need to perform a separate sorting step. The set of indices for each row of A and each column of B are already sorted. Furthermore, note that the number of non-zero entries in any row or column of A or B can vary dramatically, which is why we need a set representation that can vary as well, while still providing an efficient way to answer set-intersection queries.

To compute the Boolean product, we use perform a set-intersection query for each row i of A and each column j of B, cutting the computation short as soon as we have determined if the intersection is empty or not. Then $C[i,j] = 1$ if and only if this intersection is non-empty. By the results given above, the performance of this algorithm can be characterized as in the following theorem.

Theorem 3.2. *The product of two $n \times n$ Boolean matrices can be computed in $O(\log n)$ time and expected $O(n^2 + nm(\log w)/w)$ work in the practical CRCW PRAM model, or sequentially in $O(n^2 + nm(\log w)/w)$ expected time in the practical RAM model, where m is the number of non-zero entries in the input matrices.*

Acknowledgments. This article reports on work supported by the DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported in part from NSF grants 1228639, 1526631, 1217322, 1618301, and 1616248.

REFERENCES

[1] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster Join-projects and Sparse Matrix Multiplications. In *ICDT*. 121–126. https://doi.org/10.1145/1514894.1514909

[2] R. R. Amossen and R. Pagh. 2011. A new data layout for set intersection on GPUs. In *IPDPS*. 698–708. https://doi.org/10.1109/IPDPS.2011.71

[3] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. 1999. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comput. Sci.* 215, 1-2 (1999), 337–344. https://doi.org/10.1016/S0304-3975(98)00172-8

[4] Philip Bille, Anna Pagh, and Rasmus Pagh. 2007. Fast evaluation of union-intersection expressions. In *ISAAC (LNCS)*, Vol. 4835. 739–750. https://doi.org/10.1007/978-3-540-77120-3_64

[5] David Eppstein. 2016. Cuckoo Filter: Simplification and Analysis. In *SWAT (LIPIcs)*, Vol. 53. 8:1–8:12. https://doi.org/10.4230/LIPIcs.SWAT.2016.8

[6] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel Torres. 2017. 2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection. In *PODS*.

[7] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Vol. 17. Addison-Wesley.

[8] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2010. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* 39, 4 (2010), 1543–1561. https://doi.org/10.1137/080728743

[9] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. 2015. Dynamic Set Intersection. In *SODA*. 470–481. https://doi.org/10.1007/978-3-319-21840-3_39

[10] Andrzej Lingas. 2011. A Fast Output-Sensitive Algorithm for Boolean Matrix Multiplication. *Algorithmica* 61, 1 (2011), 36–50. https://doi.org/10.1007/s00453-010-9441-x

[11] Peter Bro Miltersen. 1996. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *ICALP (LNCS)*, Vol. 1099. 442–453. https://doi.org/10.1007/3-540-61440-0_149

[12] Mikkel Thorup. 2003. On AC^0 implementations of fusion trees and atomic heaps. In *SODA*. 699–707. http://dl.acm.org/citation.cfm?id=644108.644221