Parallel Peeling of Invertible Bloom Lookup Tables in a Constant Number of Rounds

Michael T. Goodrich[1], Ryuto Kitagawa[1], and Michael Mitzenmacher[2]

[1]Univ. of California, Irvine, [2]Harvard University

Abstract. Invertible Bloom lookup tables (IBLTs) are a compact way of probabilistically representing a set of n key-value pairs so as to support insertions, deletions, and lookups. If an IBLT is not overloaded (as a function of its size and number of key-value pairs that have been inserted), then reporting all the stored key-value pairs can also be done via a "parallel peeling" process. For the case when the IBLT is represented in a very compact form, this can be implemented to run in $O(\log \log n)$ parallel rounds, with all but inversely polynomial probability, as shown in prior work by Jiang, Mitzenmacher, and Thaler, as well as in Gao's work on parallel peeling algorithms for random hypergraphs. Although $O(\log \log n)$ is practically constant for reasonable values of n, there are nevertheless scenarios (such as in the parallel GPU or MapReduce frameworks) where parallel peeling is desired to run in a constant number of rounds, with failure probabilities that are negligible rather than simply being polynomially small. In this paper, we study simple constant-round parallel peeling algorithms for IBLTs, focusing on negligible failure probabilities based on table size, number of elements stored, and number of hash functions. We prove the surprising result that with $O(n \log n)$ space a one-round parallel peeling process succeeds with high probability while a two-round parallel peeling process succeeds with overwhelming probability. We then provide a time-space trade-off theorem for parallel peeling in a constant k number of rounds while still maintaining overwhelming success probability. We also give several new algorithmic applications of parallel peeling of IBLTs and we experimentally study the effectiveness of our approach in practice.

1 Introduction

An ***invertible Bloom lookup table*** (***IBLT***) [16,23,36] is a probabilistic hash-based data structure that concisely represents a set of key-value pairs to support insertion, deletion, lookup, and (if the IBLT is not overloaded) the listing of all the stored key-value entries. Previous work on IBLTs has focused primarily on applications of IBLTs addressing the distributed computing challenges of set reconciliation, blockchain reconciliation, and network synchronization [16,17,19,21,30,34]. In these applications, one may have a large distributed data store, and one is interested in synchronizing a set of data files or blocks across multiple servers.

Invertible Bloom Lookup Tables. When an IBLT $\mathcal{B}$ is first created, it initializes k arrays $T_1, \ldots, T_k$ of m cells, so each T_i has m/k cells. Each of the cells in a T_i table stores a constant number of fields, each of which, in turn, corresponds to a single memory word (we define these fields below), which is initially 0. An important feature of an IBLT is that at times the number of key-value pairs in $\mathcal{B}$ can be large (even larger than m), but the space used for $\mathcal{B}$ remains $O(m)$ words. The insert and delete methods never fail, whereas the listEntries method, which lists out all the key-value pairs, only guarantees good probabilistic success when the number of stored key-value pairs, n, is below an appropriate threshold. (For more details about the thresholds, see the prior works by Goodrich and Mitzenmacher [23] or Molloy [31,32].)

An IBLT uses a set of k random hash functions, h_1, h_2, ..., h_k, to determine where key-value pairs are stored. In our case, each key-value pair, (x, y), is placed into cells $T_1[h_1(x)]$, $T_2[h_2(x)]$, ... $T_k[h_k(x)]$, respectively, with fields that support all the IBLT operations. We sometimes refer to this subdivision of the IBLT into k tables as "splitting." Such splitting does not affect the asymptotic behavior in our analysis and can yield other benefits, including ease of parallelization of reads and writes into the hash table, as we show. Each cell contains the following four fields:

- count, which is a (signed) count of the number of key-value pairs that have been mapped to this cell,
- keySum, which is the XOR of every key, x, that has been mapped to this cell,
- valueSum, which is the XOR of every value, y, that has been mapped to this cell,
- hashSum, which is the XOR of a cryptographic hash, $g(x)$, of the key, x, for every key-value pair, (x, y), that has been mapped to this cell. This field is used for error-checking purposes.

Inserting a new key-value pair, (x, y), involves incrementing the count field for each cell, $T_i[h_i(x)]$, and XOR-ing into the other fields the respective parameters. Similarly, deleting a key-value pair, (x, y), involves decrementing the count field for each cell, $T_i[h_i(x)]$, and XOR-ing into the other fields the respective parameters, since every number is its own inverse under the XOR operation. For applications where we are finding symmetric set differences, we also allow for counts to become negative; that is you can delete a key-value pair that has not been inserted (as we described in the introduction). Hence a count of negative 1 can correspond to a cell that contains a single item that has been deleted. However, a count of 1 or negative 1 might correspond to multiple key-value pairs, some of which have been inserted and some of which have been deleted. The hashSum field is meant to provide an additional layer of protection to ensure that a count of 1 or negative 1 truly corresponds to a single key, by comparing the hash of the key to this field. (Note the hashSum field is not necessary if only inserting items, or only deleting items that have previously been inserted.)

The listEntries operation is more interesting. The usual way it is described is in terms of a sequential ***peeling*** process, where we look for a cell with a

– listEntries():

```
while there is an i and j such that T_i[j].count = 1 or −1 do
  for each such i and j in parallel do
    if T_i[h_i(x)].hashSum = g(T_i[h_i(x)].keySum) then
      output the pair, (T_i[j].keySum , T_i[j].valueSum)
      if T_i[j].count = 1 then
        call delete(T_i[j].keySum, T_i[j].valueSum), parallelizing the XORs
      else
        call insert(T_i[j].keySum, T_i[j].valueSum), parallelizing the XORs
```

Fig. 1: Listing entries in an IBLT. Note that in the case of a count that is -1 we actually remove its corresponding key-value pair by performing an insert.

count field that is 1 or -1, remove and report the key-value pair that is stored there, and repeat. Jiang, Mitzenmacher, and Thaler [27] study a parallel version of this peeling process, but their approach is focused on optimizing the space used, not the number of rounds, and their analysis results in a non-constant number of parallel peeling rounds. Instead, in this paper, we are interested in parameters that result in one or two or some constant, k, number of rounds of parallel peeling. For example, such approaches allow for parallel peeling to be more easily implemented in the MapReduce framework, since we can specify a specific number of peeling rounds (possibly even just one round). We describe pseudocode for a parallel peeling algorithm in Figure 1.

One of the unfortunate aspects of the pseudocode for the listEntries algorithm, from a parallel implementation standpoint, is that the outer loop is a while loop that implies a conditional number of iterations, which we call ***peeling rounds***. Implementing parallel peeling can be much easier, however, if we can simply bound the number of parallel peeling rounds to be one or two or some constant, k, and hard code that constant into our implementation, using, say, the MapReduce framework (which also simplifies combining the XORs for colliding cells). This desire, therefore, motivates the analysis that follows, which shows how to set the parameters for an IBLT to guarantee with high probability (or all but negligible probability)[1] that parallel peeling can succeed in one, two, or k rounds.

Set Synchronization. For example, using IBLTs allows one to synchronize two data sets, S_1 and S_2, using storage and communication proportional to the size of symmetric set difference between the two sets, $n = |S_1 \oplus S_2|$, rather than the size of the sets themselves.[2]

At a high level, synchronizing a set S that is intended to be mirrored at two servers, which we call "Alice" and "Bob," works as follows. Alice and Bob hold

[1] We say that an event occurs with high probability if the failure probability is $1/n^c$, for some constant, $c > 0$, and with overwhelming probability if the failure probability is negligible, that is, the failure probability is asymptotically less than $1/n^c$ for any constant $c > 0$.

[2] We denote the symmetric set difference of two sets, S_1 and S_2, by $S_1 \oplus S_2$. Recall that $S_1 \oplus S_2 = (S_1 - S_2) \cup (S_2 - S_1)$, which is sometimes alternatively denoted as $S_1 \ominus S_2$ or $S_1 \Delta S_2$.

item sets S_A and S_B, respectively, with $S = S_A \cup S_B$. Alice and Bob each store their items in respective IBLTs, T_A and T_B, each holding m cells, where m is linear in the size of the set difference $S_A \oplus S_B$, and each sends their IBLT to the other. Then, each of Alice and Bob computes an entrywise table difference $T_A - T_B$, which allows both Alice and Bob to list out the elements of $S_A \oplus S_B$ with high probability, so long as the size of that difference is indeed at most a threshold that depends only on and is linear in m. Note that computing the table difference also yields which set (Alice's or Bob's) each element in the symmetric difference belongs to; we therefore say that IBLTs provide *signed* symmetric set differences. While this is an interesting and useful application of IBLTs, in this paper we are interested in applications of IBLTs to another challenge that arises in large-scale distributed computing, namely, for data compression [38] and deduplication [44], which can involve using IBLTs in arguably a more "parallel" way. For instance, in data compression and deduplication applications, one is often interested in computing the symmetric or set differences between all pairs of a collection of sets, not just two. Thus, we desire IBLT difference operations that can be done quickly in parallel, where each difference itself is computed using a parallel algorithm, ideally with a constant number of computation rounds.

Prior Work on Parallel Peeling. Computing the entrywise table difference $T_A - T_B$ between two IBLTs T_A and T_B is easy enough to do in parallel, but listing out the elements of the result in parallel is more challenging. Indeed, the algorithm for performing such a listing is usually described as a sequential ***peeling*** process, where items are removed iteratively one at a time [23,36]. Still, there is prior work on ***parallel peeling*** of various graph structures, e.g., by Cao, Fineman, and Russell [10], Goodrich and Pszona [24], Chang, Pettie, and Zhang [11], Dhulipala, Blelloch, and Shun [14], Ghaffari, Grunau, and Jin [22], Shi, Dhulipala, and Shun [39], and Shi and Shun [40]. More relevant to this paper, Jiang, Mitzenmacher, and Thaler [27] and Gao [20] provide parallel peeling results for random hypergraphs and IBLTs. These results yield a super-constant number of rounds, however, rather than a small constant number of rounds, as would be desired for the applications we explore here. Specifically, to peel n items in the IBLT, their results correspond to $\Theta(\log \log n)$ rounds of parallel peeling.

Our Results. In this paper, we study constant-round parallel peeling both theoretically and experimentally. What makes our study of the IBLT data structure different from prior parallel-peeling results is that here we focus more on optimizing (parallel) time and work and less on space, which was the focus in prior works [20,27]. In particular, we guarantee that with high (and, as we describe later, with overwhelming) probability our parallel peeling process completes in a small constant number of rounds—ideally one or two. Past work on parallel peeling for IBLTs [20,27], instead aimed for a constant number of hash functions per item, a linear number of cells in the IBLT table, and an asymptotically good, but super-constant, number of rounds (where additional $O(1)$ terms were not considered consequential) with high, but not overwhelming probability. On the theoretical front, we show that if an IBLT has $O(n \log n)$ cells and uses $O(\log n)$ hash functions (where suitably chosen constant factors

are used in the order notation), then, with high probability, we can peel the IBLT in a single parallel round. More surprisingly, we also show that the probability that we would fail to peel such an IBLT in only two rounds is negligible (recall that a function is ***negligible*** if it approaches zero faster than the reciprocal of any polynomial, e.g., see Bellare [4]). Further, we provide a full time-space trade-off between the number of rounds and space needed to achieve constant-round parallel peeling with overwhelming probability. Also, we explicitly describe how to use IBLTs in some interesting applications, including deduplication and symmetric-difference minimum spanning trees.

2 Analysis

In this section, we provide our theoretical analysis, showing that, for a table of $m = O(n \log n)$ cells, parallel peeling succeeds with high probability in one round and with overwhelming probability in two, and we then provide a time-space trade-off while achieving overwhelming probability with a constant number of rounds. We note that in the analysis in this section, we assume that there is no false positive in the peeling process in checking the hashSum field; that is, we assume items have only been inserted, or the fingerprints have been chosen large enough to avoid this issue. Before beginning, we recall that previous work (e.g., [23]) has shown for an IBLT with $m = cn$ cells for some constant c and a constant number of hash functions k, if c is above the threshold where decoding occurs with high probability, the failure probability is $\Theta(n^{-k+2})$.

Analysis of One and Two Rounds. As we are focused on such a small number of rounds, we first explicitly consider the case of a single round and then analyze the probability of parallel peeling succeeding in two rounds. We note that something akin to our analysis for one round appeared previously in work by Eppstein and Goodrich [15], but we nevertheless provide a complete analysis of one round parallel peeling to set the table for our other results, which are novel.

For concreteness, we consider an IBLT with $m = cn \log_2 n$ cells for some constant c. Each of n items hashes to $\log_2 n$ cells (we assume n is a power of two for convenience). As we previously described, we assume the IBLT is ***split***; that is, there are $\log_2 n$ subtables, each with cn cells, and the jth hash of each item is independently and uniformly chosen in the jth subtable. Call a cell ***single*** if it holds exactly one item. We show for sufficiently large constant c each item has at least one hash to a single cell with high probability, implying peelability in a single round.

Theorem 1. *For a table of $m = cn \log_2 n$ cells using $\log_2 n$ hash functions, where $c \geq 1$ is a constant, an IBLT peels in one round with probability at least $1 - n^{-c'+1}$ where $c' = -\log_2(1 - e^{-1/c})$.*

Proof: Let $X_{i,j}$ be a random variable such that $X_{i,j} = 1$ if the jth hash of the ith item is not single and 0 otherwise. Since the jth table has cn cells, we have

$$\Pr(X_{i,j} = 0) = (1 - 1/(cn))^{n-1} \geq e^{-1/c}.$$

(The inequality holds as long as $c \geq 1$ for example.) Since subtables are independent, we therefore have the probability that no hash for the ith item is single is at most

$$(1 - e^{-1/c})^{\log_2 n} = n^{\log_2(1-e^{-1/c})} = n^{-c'},$$

for $c' = -\log_2(1-e^{-1/c})$. By a union bound, all items hash to at least one single cell with probability $n^{-c'+1}$, and one can choose c to obtain any suitably small probability that is inversely polynomial in n. ■

Thus, in our setup, parallel peeling succeeds in one round with high probability and with reasonable constant factors. For example, one can peel in a single round with probability at least $1 - 1/n$ using slightly less than $3.5n \log_2 n$ cells, and with probability at least $1 - 1/n^2$ with slightly less than $7.5n \log_2 n$ cells. Further, we expect a threshold where the probability that one round suffices jumps toward 1 at slightly over $1.443n \log_2 n$ cells, since $1.443 \approx 1/\ln 2$ and at $c = 1/\ln 2$ we have $c' = 1$.

We note the above proof doesn't really require $c \geq 1$, in that for smaller c we have that $(1-1/(cn))^{n-1} = e^{-1/c}(1-o(1))$ and the proof remains essentially the same. This fact will be helpful in what follows. Also, the proof naturally extends to other scenarios, such as if one uses $\alpha \log n$ hash functions for some constant α (or some larger number of hash functions).

We now consider just the case of two rounds. We use the same setup as before. For an item to be peeled within two rounds, either

- one of its cells is single, or
- one of its cells has that all the other items that hash to that cell are peeled after the first round.

We prove that with just two rounds the failure probability is negligible; specifically, the probability of failing to peel every item is $n^{-\Omega(\log n)}$.

Theorem 2. *For a table of $m = cn \log_2 n$ cells using $\log_2 n$ hash functions, where $c > 0$ is a constant, an IBLT peels in two rounds with probability at least $1 - n^{-\Omega(\log n)}$.*

Proof: Consider a specific item y. Let Y_j be the number of other items that hash to the same cell as y in the jth subtable. We first note that Y_j is at most $(\ln n)^2$ with probability at least $1 - n^{-\Omega(\log n)}$. This follows readily from the Poisson approximation of the number of items that hash to each cell, along with Stirling's approximation. We therefore assume that every cell has at most $(\ln n)^2$ items that hash to it henceforth, as this conditioning does not affect our arguments further. Let $X_i = 1$ if y is single in the ith table or all other items that share the cell in the ith table all peel after the first round. Otherwise $X_i = 0$. We wish to show that the probability that all $X_i = 0$ is $n^{-\Omega(\log n)}$.

First consider X_1. Let z be some other item in the same cell. Following the same argument as in the proof of Theorem 1, the probability that z is not single in at least one of the other subtables is

$$\left(1 - (1 - 1/(cn))^{n-1}\right)^{\log_2 n - 1},$$

which is at most n^{-c_1}, for some constant c_1. As we have assumed Y_1 is at most $(\ln n)^2$, by a union bound, the probability any of the items is not single in at least one other subtable is at most $(\ln n)^2 n^{-c_1}$, and this shows the probability $X_1 = 0$ is at most $(\ln n)^2 n^{-c_1}$.

If the X_i were all independent, we would now be done. However, the X_i are only "roughly independent"; there are unfortunately some problematic cases to consider. For example, consider $\Pr(X_2 = 1 \mid X_1 = 0)$. That is, what is the probability the second subtable allows us to peel the item y even though the first subtable does not. A difficult subcase showing the dependency is when the same second item hashes to the same location as y in both hash tables. That is, the reason $X_1 = 0$ might be that there is an item z in the second hash table at the same location as y in both the first and second table.

We circumvent the dependency by showing the following two conditions hold:

1. With probability $1 - n^{-\Omega(\log n)}$, at least $(\log_2 n)/2$ subtables have no items in the cell with y that also appear with y in earlier subtables.
2. For each such subtable, the event $X_i = 0$ satisfies $\Pr(X_i = 0 \mid X_1 = 0, X_2 = 0, \ldots, X_{i-1} = 0) = O(n^{-c_2})$, for some constant c_2.

The result would then follow, as (implicitly conditioning on none of the rare $n^{-\Omega(\log n)}$ events we have considered occurring)

$$\Pr(X_1 = 0, X_2 = 0, \ldots, X_{\log_2 n} = 0) =$$

$$\Pr(X_1 = 0) \prod_{i=2}^{\log_2 n} \Pr(X_i = 0 \mid X_1 = 0, X_2 = 0, \ldots, X_{i-1} = 0),$$

and the right hand side would have at least $(\log_2 n)/2$ terms that were $O(n^{-c_2})$.

For the first condition, as we sequentially consider each new subtable, there are at most $O((\log n)^3)$ items that share a cell with y. The probability that in a new subtable any of these elements are in the same cell as y is at most q, where q is $O((\log n)^3)/n$. The probability at least $(\log_2 n)/2$ subtables would fail to avoid such elements is at most

$$\sum_{i=(\log_2 n)/2}^{\log_2 n} \binom{\log_2 n}{i} q^i (1-q)^{\log_2 n - i},$$

which is $n^{-\Omega(\log n)}$.

For the second condition, following previous work, it is useful to think of gathering a history of all the item-cell pairs we have seen as we explore the hash table subtable by subtable. We start with y and its position in all the tables. In the first subtable, we consider all the items that share the cell with y; we then consider the cells for those items in all the other tables (to check if those other items are single in some table, and are hence peeled in the first round), and correspondingly all the items in those cells (which, because we have now seen them, we must consider their effect on the conditioning). Similarly, at the second table, we consider all the items that share the cell with y; we then consider the cells for those items in all the other tables, and correspondingly all the items in those cells. Under our assumption that any cell contains at most $(\ln n)^2$ items, we see that this history can only consist of $O((\log n)^6)$ item-cell pairs throughout the process.

Now consider $\Pr(X_i = 0 \mid X_1 = 0, X_2 = 0, \ldots, X_{i-1} = 0)$ for some subtable i where the items colliding with y are all distinct from other levels. The past history introduces some conditioning in evaluating whether each of these items is single in another subtable, because we know the location of some items in other tables in our history. However, we only know at most $O((\log n)^6)$ item-cell pairs in our history. This has at worst a $1-o(1)$ effect on the probability $(1-1/(cn))^{n-1}$ that an item colliding with y in the ith table is single in another table, as from the calculation in the proof of Theorem 1. In particular, any such new item that collides with y must avoid cells known to contain other items in other subtables. As we have said, there are polylog(n) such cells from our history, so this happens with probability $1 - \text{polylog}(n)/(cn)$ in any given subtable. And it can affect the number of other items that might collide with the new item (instead of $n-1$ other items may be $n - \text{polylog}(n)$, as polylog(n) items might already their position known in a subtable). It remains the case that any new item is single in each subtable with probability $(1 - \text{polylog}(n)/(cn))^{n-\text{polylog}(n)} = e^{-1/c}(1 - o(1))$, and correspondingly each new item is single in some other subtable with probability at least $1 - n^{-c_3}$, for some constant c_3. The probability that all new items in the ith subtable are not single is then $\text{polylog}(n)/n^{c_3} = O(n^{-c_2})$ for some constant c_2, giving the result. ■

A Time-Space Trade-off for Constant-Round Peeling. Theorem 2 shows parallel peeling succeeds in two rounds with overwhelming probability. Let us next provide our time-space trade-off.

Theorem 3. *For a table of $m = cn(\log_2 n)^{1/k}$ cells using $(\log_2 n)^{1/k}$ hash functions, where where k is a positive integer constant and $c > 0$ is a constant, an IBLT peels in $k+1$ rounds with probability at least $1 - n^{-\Omega((\log n)^{1/k})}$.*

Proof: We sketch the proof, since it follows roughly the same conceptual framework as the proof of Theorem 2. Consider a specific item, y. For y not to be peeled in $k+1$ rounds, in the ith subtable there must be some element z_i that has not been peeled after k rounds in the same cell as y; for each z_i, in each of the other subtables there must be some element that has not been peeled after $k-1$ rounds in the same cell as z_i, and so on.

Accordingly, the failure of an IBLT to peel corresponds to what is commonly referred to as a witness tree (e.g., [41]), with the root of the tree being an element y not peeled after $k+1$ rounds, connected by labeled edges to children that correspond to elements that share a cell with y in some subtable that are not peeled after k rounds, with the label denoting which subtable y and the other element share a cell. (An element can conceivably be in more than one labeled edge.) These elements have children corresponding to elements that share a cell with them in some other subtable (other than the one in the edge connecting them to the root) that are not peeled after $k-1$ rounds, and so on.

We can again use the fact that there are at most $(\ln n)^2$ elements in any cell with probability at least $1-n^{-\Omega(\log n)}$ to limit the size of the tree built this way, so that with overwhelming probability the tree has size polylogarithmic in n.

We think of going through this tree in a breadth first manner from the root, and temporarily assume that elements are not repeated as we expand this recursive exploration of the table and likewise ignore dependencies introduced by knowledge of where elements we have seen in the tree are placed. The probability that an element q is not single in at least one of $(\log_2 n)^{1/k}-1$ subtables is

$$\left(1-(1-1/(cn))^{n-1}\right)^{(\log_2 n)^{1/k}-1},$$

which is at most $c_1^{(\log_2 n)^{1/k}}$, for some constant $c_1<1$ and sufficiently large n. The probability that an element q' in a cell is not peeled after two rounds, implying that in all of the other $(\log_2 n)^{1/k}-1$ subtables there is some element in the same cell as q' that is not peeled after 1 round, is at most

$$(\ln n)^2(\log_2 n)^{1/k}\left(c_1^{(\log_2 n)^{1/k}}\right)^{(\log_2 n)^{1/k}-1} \le c_2^{(\log_2 n)^{2/k}},$$

for some constant $c_2<1$ and sufficiently large n. Continuing in this manner, the probability an element sharing a cell with y in one of its subtables has not been peeled for sufficiently large n is bounded by

$$c_k^{(\log_2 n)^{k/k}} = n^{\log_2 c_k},$$

where $c_k<1$. The result would correspondingly follow, but for the assumptions that elements in the table are not repeated and that dependencies can be ignored.

As in the proof of Theorem 2, however, because the witness tree is only polylogarithmic in size, this does not affect the asymptotics of the result; the dependencies can be dealt with by using the fact that we only see a polylogarithmic number of elements throughout the tree, and most of the cells will not contain repeated elements from the tree. ■

3 Applications

In this section, we describe some applications of constant-round parallel peeling algorithms for IBLTs to a simplified deduplication problem.

All-pairs Signed Symmetric Set Differences. We begin by describing how to use an IBLT for computing the (signed) symmetric set differences between every pair of a collection of N sets, $S_1, S_2, \ldots, S_N$, by adapting an approach of Goodrich and Mitzenmacher [23] to our framework. (Note this is equivalent to finding the set difference between every pair of sets.) Let $n > 1$ be an upper bound for the anticipated size of any difference, $S_i \oplus S_j$. (Note that n can be much smaller than the size of any S_i.) We begin by computing an IBLT, T_{S_i}, for each set, S_i, of $O(n \log n)$ cells, and with $O(\log n)$ hash functions (and subtables), based on the theoretical and/or experimental analysis we provide above. In this case, we store each element, x, from a set, S_i, as a key-value pair, $(x, 1)$, since we are treating the S_i's as sets.

For each pair, (i, j), where $i \neq j$, $i, j = 1, 2, \ldots, N$, we compute a set-difference IBLT, $T_{i,j}$, by computing an indexed difference of the corresponding count fields in T_{S_i} and T_{S_j} and an indexed XOR of the corresponding keySum, valueSum, and hashSum fields. We emphasize that in the results of this section, we assume that cell operations (such as XORing the various fields of a cell) are unit cost, even if the fields are large. (As we have noted, in theory the hashSum field may need to be $\omega(\log n)$ bits for some applications and/or for negligible failure probability; in practice, we expect all fields to fit in one or a small constant number of machine words.) Thus, $T_{i,j}$ is a representation of the signed symmetric set difference,[3] $S_i \oplus S_j$, where each element, x, that is in S_i and not in S_j, adds 1 to its respective count fields, and each element, x, that is in S_j and not in S_i, adds -1 to its respective count fields. We then apply the method described in Section 1 to perform a parallel peeling to list of the elements in $T_{i,j}$, optionally noting which elements in the difference are from S_i and which are from S_j.

Theorem 4. *Given N sets, $S_1, S_2, \ldots, S_N$, if the size of the difference between two sets, S_i and S_j, is at most a given size parameter, $n > 1$, then the probability that our algorithm will fail to compute the signed set difference between S_i and S_j in at most two rounds is negligible as a function of n. The total work needed is $O(NM \log n + N^2 n \log n)$.*

Proof: The work bound follows from the work need to insert each element into its set's IBLT and then perform all the set differences. The probability bounds follow by Theorem 2. ■

Deduplication via Difference Encodings. Consider now a simple deduplication [44] problem. Suppose we are given a collection of sets, $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$. For example, each S_i could represent a file or a database, and its elements could be disk blocks, database rows, or identifiers for disk blocks or database rows derived from a cryptographic hash function, such as SHA-256. In deduplication applications, it is anticipated that there are a lot of common elements among

[3] Recall that we say that $S_i \oplus S_j$ is a ***signed symmetric difference*** if we can separately identify the members of $S_i - S_j$ and $S_j - S_i$, that is, the members of S_i not in S_j and the members of S_j not in S_i.

the sets in $\mathcal{S}$; hence, we would like to represent each S_i concisely, e.g., just in terms of a small number of differences with another set in $\mathcal{S}$.

One such representation is a ***difference encoding*** of $\mathcal{S}$, which is a (re)ordering of the sets in $\mathcal{S}$, as $(S_1, S_2, \ldots, S_N)$, such that, for $i > 1$, each set, S_i, is encoded in terms of the differences between S_i and some S_j where $j < i$. That is, we encode S_i with the signed symmetric difference, $S_i \oplus S_j$. Thus, if there is considerable overlap of elements among the sets in $\mathcal{S}$, then a difference encoding could save a lot of memory space over explicitly representing each set in $\mathcal{S}$. The corresponding optimization problem, therefore, is to find an ordering of the sets and a difference encoding that minimizes the total amount of storage required, over all possible orderings and difference encodings. Finding such a ***minimum difference encoding*** might at first seem like a computationally difficult problem, e.g., since there are $N!$ possible orders and each one can have $(N-1)!$ difference encodings. Nevertheless, as we show below, we can solve this problem quickly in parallel with reasonable work.

Symmetric-difference MSTs. As has been well-known since at least the 1950s, e.g., see Restle [37], given a collection of sets, $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$, and a measure function, m, for sets (such as their size if all the sets are finite), then $d(S_i, S_j) = m(S_i \oplus S_j)$ is a distance metric. In our case, since the sets we are dealing with are finite, we define $m(S)$ to be the size of the set, S, which we denote as $|S|$.

Define a graph, G, which we call the ***symmetric-difference graph***, such that each set, S_i, in $\mathcal{S}$ is associated with a vertex, i, in G, and each edge, (i, j), has weight equal to $|S_i \oplus S_j|$.

Lemma 1. *Finding a minimum difference encoding for a collection of sets, $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$, can be reduced to finding a minimum spanning tree (MST) in the symmetric-difference graph, G, for $\mathcal{S}$.*

Proof: Let H be a spanning tree of G. We can derive a difference encoding of $\mathcal{S}$ from H by choosing the string associated with a vertex in H (it doesn't matter which one) to be the first string in the order. We then root the tree H at this first vertex and order the remaining vertices according to a preorder traversal of H, and let this be the ordering of the corresponding strings. In this way, we are guaranteed that for each vertex, i, its parent in H appears earlier in the ordering. We then encode every vertex, i, besides the first one in terms of the symmetric difference between S_i and S_j, where j is the parent of i in H. Thus, H corresponds to a difference encoding of $\mathcal{S}$.

Alternatively, let D be a difference encoding of $\mathcal{S}$ and let the corresponding order be $(S_1, S_2, \ldots, S_N)$. For each set, S_i, for $i > 1$, choose the (directed) edge, (i, j), in G such that S_i is encoded as a difference with S_j, and let H be the resulting subgraph of G. Then H has no cycles, because each vertex, $i > 1$, we chose an out-going edge to a vertex for j where $j < i$. In addition, for the same reason, H is connected, with each directed path leading to the first vertex (for S_1). Thus, H is a spanning tree, with $n - 1$ edges.

Therefore, every spanning tree corresponds to a difference encoding and every difference encoding corresponds to a spanning tree; hence, an MST for G will correspond to a minimum difference encoding for $\mathcal{S}$. ■

A major computational bottleneck for solving the above symmetric-difference MST problem is in constructing a representation of the symmetric-difference graph, G. Of course, since we are interested in finding an MST in G, it is sufficient to construct a representation of G such that every edge has weight at most some size threshold, $n > 0$, while keeping G connected. Typically, we desire n to be much smaller than M, the average size of each set in $\mathcal{S}$.

Our method for constructing G is to use IBLTs and our parallel peeling algorithm. Namely, we use our algorithm for the all-pairs set difference problem. We define a reasonable threshold, n, for the maximum expected symmetric difference so that G is connected, and run our all-pairs set difference algorithm of Theorem 4. Note that if our threshold, n, is large enough, then the probability that any of our parallel peeling algorithms fail after two rounds is negligible in n, by Theorem 2; hence, if our peeling algorithm fails for some pair (S_i, S_j), we can safely assume that $|S_i \oplus S_j| > n$. Thus, if after running our all-pairs set difference algorithm, this results in a graph, G, that is not connected, then we double our estimate for n and run it again. Since we double the value of n in each such run and our work bound is at least linear in n, the work needed the runs forms a geometric sequence that is dominated by the work for the last run. This gives us the following:

Theorem 5. *Given a collection, $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$, of N sets, with average size, M, we can determine a threshold value, n, and we can construct a connected subgraph of the symmetric-difference graph, G, for $\mathcal{S}$, containing every edge with weight at most n in $O(\log n)$ rounds in parallel, with total work $O(NM \log n + N^2 n \log n)$, with a failure probability that is negligible in n.*

Given such a connected subgraph of the symmetric-difference graph, G, we can then compute an MST in G using any known parallel MST algorithm, e.g., see [2,5,13,28]. Note that the above method also applies to a set, $S = \{s_1, s_2, \ldots, s_N\}$, of N character strings of length, $M \geq 1$, each, since we can construct a set, S_i, from each character string, s_i, by defining each element in S_i to be the pair, $(k, s_i[k])$. In this case, the symmetric-difference graph, G, would be defined so that each string s_i corresponds to a vertex and the weight of an edge, (i, j), is the Hamming distance[4] between the strings s_i and s_j. This approach could be used, for example, to concisely encode a set of DNA strings where differences are character swaps or replacements (but not insertions or deletions). In this case, the minimum difference encoding would provide an optimized concise encoding of all the strings in S.

[4] Recall that the Hamming distance between two strings, s and t, is the number of positions, i, where $s[i] \neq t[i]$.

References

1. Adhikari, V.K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., Zhang, Z.L.: Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In: IEEE INFOCOM. pp. 1620–1628 (2012). https://doi.org/10.1109/INFCOM.2012.6195531
2. Adler, M., Dittrich, W., Juurlink, B., Kutyłowski, M., Rieping, I.: Communication-optimal parallel minimum spanning tree algorithms. In: 10th ACM Symp. on Parallel Algorithms and Architectures (SPAA). pp. 27–36 (1998)
3. B. Jenkins: A hash function for hash table lookup. https://burtleburtle.net/bob/hash/doobs.html (1997)
4. Bellare, M.: A note on negligible functions. Journal of Cryptology **15**(4) (2002)
5. Bentley, J.L.: A parallel algorithm for constructing minimum spanning trees. Journal of Algorithms **1**(1), 51–59 (1980). https://doi.org/10.1016/0196-6774(80)90004-8
6. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
7. Böttger, T., Cuadrado, F., Tyson, G., Castro, I., Uhlig, S.: Open Connect everywhere: A glimpse at the Internet ecosystem through the lens of the Netflix CDN. SIGCOMM Comput. Commun. Rev. **48**(1), 28–34 (apr 2018). https://doi.org/10.1145/3211852.3211857, doi.org/10.1145/3211852.3211857
8. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. Internet Mathematics **1**(4), 485–509 (2004)
9. Brodtkorb, A.R., Hagen, T.R., Schulz, C., Hasle, G.: GPU computing in discrete optimization. Part I: Introduction to the GPU. EURO Journal on Transportation and Logistics **2**(1), 129–157 (2013). https://doi.org/https://doi.org/10.1007/s13676-013-0025-1, https://www.sciencedirect.com/science/article/pii/S2192437620600267
10. Cao, N., Fineman, J.T., Russell, K.: Parallel shortest paths with negative edge weights. In: 34th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). pp. 177–190 (2022)
11. Chang, Y.J., Pettie, S., Zhang, H.: Distributed triangle detection via expander decomposition. In: 13th ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 821–840. SIAM (2019)
12. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
13. Dehne, F., Gotz, S.: Practical parallel algorithms for minimum spanning trees. In: 17th IEEE Symposium on Reliable Distributed Systems. pp. 366–371 (1998). https://doi.org/10.1109/RELDIS.1998.740525
14. Dhulipala, L., Blelloch, G., Shun, J.: Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In: 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). pp. 293–304 (2017)
15. Eppstein, D., Goodrich, M.T.: Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. IEEE Transactions on Knowledge and Data Engineering (to appear)
16. Eppstein, D., Goodrich, M.T.: Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. IEEE Transactions on Knowledge and Data Engineering **23**(2), 297–306 (2010)
17. Eppstein, D., Goodrich, M.T., Uyeda, F., Varghese, G.: What's the difference? efficient set reconciliation without prior context. ACM SIGCOMM Computer Communication Review **41**(4), 218–229 (2011)

18. Fagerjord, A., Kueng, L.: Mapping the core actors and flows in streaming video services: What Netflix can tell us about these new media networks. Journal of Media Business Studies **16**(3), 166–181 (2019). https://doi.org/10.1080/16522354.2019.1684717, doi.org/10.1080/16522354.2019.1684717
19. Fu, W., Abraham, H.B., Crowley, P.: Synchronizing namespaces with invertible Bloom filters. In: ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS). pp. 123–134 (2015). https://doi.org/10.1109/ANCS.2015.7110126
20. Gao, P.: Analysis of the parallel peeling algorithm: A short proof (2014), arxiv.org/abs/1402.7326
21. Gentili, M.: Set Reconciliation and File Synchronization Using Invertible Bloom Lookup Tables. Ph.D. thesis, Harvard Univ. (2015)
22. Ghaffari, M., Grunau, C., Jin, C.: Improved MPC algorithms for MIS, matching, and coloring on trees and beyond. In: 34th International Symposium on Distributed Computing. pp. 34:1–34:18 (2020)
23. Goodrich, M.T., Mitzenmacher, M.: Invertible Bloom lookup tables. In: 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton). pp. 792–799. IEEE (2011), arxiv.org/abs/1101.2245
24. Goodrich, M.T., Pszona, P.: External-memory network analysis algorithms for naturally sparse graphs. In: Demetrescu, C., Halldórsson, M.M. (eds.) Algorithms – ESA 2011. pp. 664–676. Springer, Berlin, Heidelberg (2011)
25. Hijma, P., Heldens, S., Sclocco, A., van Werkhoven, B., Bal, H.E.: Optimization techniques for GPU programming. ACM Comput. Surv. **55**(11) (2023). https://doi.org/10.1145/3570638, https://doi.org/10.1145/3570638
26. Hussain, A., Aleem, M.: GoCJ: Google cloud jobs dataset for distributed and cloud computing infrastructures. Data **3**(4), 38 (2018)
27. Jiang, J., Mitzenmacher, M., Thaler, J.: Parallel peeling algorithms. ACM Trans. Parallel Comput. **3**(1) (Jan 2017). https://doi.org/10.1145/2938412, doi.org/10.1145/2938412
28. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 938–948 (2010). https://doi.org/10.1137/1.9781611973075.76, https://epubs.siam.org/doi/abs/10.1137/1.9781611973075.76
29. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for MapReduce. In: 21st ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 938–948 (2010)
30. Mizrahi, A., Bar-Lev, D., Yaakobi, E., Rottenstreich, O.: Invertible Bloom lookup tables with listing guarantees. arXiv:2212.13812 (2022)
31. Molloy, M.: The pure literal rule threshold and cores in random hypergraphs. In: 15th ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 672–681. Society for Industrial and Applied Mathematics (2004)
32. Molloy, M.: Cores in random hypergraphs and boolean formulas. Random Structures & Algorithms **27**(1), 124–135 (2005)
33. Odun-Ayo, I., Ajayi, O., Akanle, B., Ahuja, R.: An overview of data storage in cloud computing. In: IEEE Int. Conf. on Next Generation Computing and Information Systems (ICNGCIS). pp. 29–34. IEEE (2017)
34. Ozisik, A.P., Andresen, G., Levine, B.N., Tapp, D., Bissias, G., Katkuri, S.: Graphene: Efficient interactive set reconciliation applied to blockchain propagation. In: ACM SIGCOMM. p. 303–317 (2019). https://doi.org/10.1145/3341302.3342082, doi.org/10.1145/3341302.3342082

35. Palankar, M.R., Iamnitchi, A., Ripeanu, M., Garfinkel, S.: Amazon S3 for science grids: A viable solution? In: ACM Int. Workshop on Data-Aware Distributed Computing. p. 55–64 (2008). https://doi.org/10.1145/1383519.1383526, doi.org/10.1145/1383519.1383526
36. Pontarelli, S., Reviriego, P., Mitzenmacher, M.: Improving the performance of invertible Bloom lookup tables. Information Processing Letters **114**(4), 185–191 (2014)
37. Restle, F.: A metric and an ordering on sets. Psychometrika **24**(3), 207–220 (1959)
38. Sayood, K.: Introduction to Data Compression. Morgan Kaufmann (2017)
39. Shi, J., Dhulipala, L., Shun, J.: Parallel clique counting and peeling algorithms. In: SIAM Conf. on Applied and Computational Discrete Algorithms (ACDA). pp. 135–146 (2021). https://doi.org/10.1137/1.9781611976830.13, https://epubs.siam.org/doi/abs/10.1137/1.9781611976830.13
40. Shi, J., Shun, J.: Parallel algorithms for butterfly computations. In: Symposium on Algorithmic Principles of Computer Systems (APOCS). pp. 16–30 (2020). https://doi.org/10.1137/1.9781611976021.2, https://epubs.siam.org/doi/abs/10.1137/1.9781611976021.2
41. Vöcking, B.: How asymmetry helps load balancing. J. ACM **50**(4), 568–589 (2003). https://doi.org/10.1145/792538.792546, https://doi.org/10.1145/792538.792546
42. Wilder, B.: Cloud Architecture Patterns: Using Microsoft Azure. "O'Reilly Media, Inc." (2012)
43. Wittig, M., Wittig, A.: Amazon Web Services in Action. Simon and Schuster (2018)
44. Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y., Zhou, Y.: A comprehensive study of the past, present, and future of data deduplication. Proceedings of the IEEE **104**(9), 1681–1710 (2016)