Parallel Bit-Counting for Approximate Similarity Searching

Rasmus Hag Løvstad

Block 1+2, 2021

Abstract

This project regards an analysis of recent advances in solving the Approximate Jaccard Similarity Search Problem, specifically in regards to how one can achieve sublinear query time using parallel bit counting as presented by Knudsen[10]. This contains a theoretical analysis of both runtime and correctness of the bit-counting algorithm as well as an empirical comparison to existing methods. The findings include both a theoretical and empirical run time advantage to using parallel bit counting compared to a simple, linear time algorithm. Furthermore, reflections are made on how the results might scale on more specialized hardware.

Contents

1 Introduction **3**

2 Theory **4**
- 2.1 Problem Definition . . . 4
- 2.2 Trivial Solution . . . 4
- 2.3 Locality Sensitive Hashing Framework . . . 5
- 2.4 MinHash . . . 6
  - 2.4.1 Implementation with the LSH framework . . . 7
- 2.5 Fast Similarity Sketching . . . 8
- 2.6 Parallel Bit Counting . . . 9
  - 2.6.1 Divide-and-Conquer . . . 10
  - 2.6.2 Parallelism . . . 11

3 Methods **15**
3.1 Hypothesis 15
3.2 Benchmarking 15

4 Implementation **17**
4.1 Technology 17
4.2 Design 17
4.3 Correctness 17
4.4 Benchmarking 18

5 Results **19**

6 Discussion **20**

7 Conclusion **21**

A Machine Specifications **22**

1 Introduction

The *Approximate Similarity Search Problem* regards efficiently finding a set A from a corpus $\mathcal{F}$ that is approximately similar to a query set Q in regards to the *Jaccard Similarity* metric $J(A, Q) = \frac{|A \cap Q|}{|A \cup Q|}$[6][10]. Practical applications includes searching through large corpi of high-dimensional text documents like plagiarism-detection or website duplication checking among others[8]. The main bottleneck in this problem is the *curse of dimensionality.* Any trivial algorithm can solve this problem in $O(nd|Q|)$ time, but algorithms that query in linear time to the dimensionality of the corpus scale poorly when working with high-dimensional datasets. Text documents are especially bad in this regard since they often are encoded using *w-shingles* (w contigous words) which Li, Shrivastava, Moore, *et al.* [5] shows easily can reach a dimensionality upwards of $d = 2^{83}$ using just 5-shingles.
The classic solution to this problem is the MinHash algorithm presented by Broder [1] to perform website duplication checking for the AltaVista search engine. It preprocesses the data once using hashing to perform effective querying in $O(n + |Q|)$ time, a significant improvement independent of the dimensionality of the corpus. Many improvements have since been presented to both improve processing time, query time and space efficiency. Notable mentions includes (but are not limited to) *b-bit minwise hashing*[4], *fast similarity sketching*[6] and *parallel bit-counting*[10] (the latter of which is the main focus of this project). These contributions have brought the query time down to sublinear time while keeping a constant error probability.
The addition of parallel bit-counting for querying

2 Theory

The Similarity Search Problem has many variations and uses. The general idea is to find some data point from a set of datapoints that is the most similar to some query point according to some similarity metric. The LSH framework, one of the first big breakthroughs in this field, regards points in a d-dimensional space and uses the Manhattan distance between them as a similarity metric - two points that are close together are regarded as more similar[3]. In this project, we will regard the similarity between sets and use the Jaccard similarity $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ as our similarity metric. The Jaccard similarity produces a number between 0 and 1, where a larger number indicates more similarity between the two sets.
This part will first define the problem to be considered in a concise way - then describe the

2.1 Problem Definition

The formal definition of the *Jaccard Similarity Search Problem* is as follows: Given a family $\mathcal{F}$ of n sets from some universe U and a query set Q, preprocess $\mathcal{F}$ to efficiently find the set $S \in \mathcal{F}$ such that $S = \max_{A \in \mathcal{F}} J(A, Q)$.
In many practical applications, it may be enough to only consider the *Approximate Jaccard Similarity Search Problem* (as defined by Knudsen [10]): Given a family $\mathcal{F}$ from a universe U, a query set Q and two variables $0 < j_2 < j_1 < 1$, preprocess $\mathcal{F}$ such that one can efficiently find some set $A \in \mathcal{F}$ such that $J(A, Q) \geq j_2$ if there exists some set $B \in \mathcal{F}$ where $J(B, Q) \geq j_1$. Note that it is possible for $A = B$. The intuition to this is that we can dial in how precise we expect our algorithm to be able to solve this problem by fine-tuning j_2 and j_1. We no longer expect the algorithm to return the *best* solution, just one that is "good enough". Algorithms that solve this problem should also be able to return "no answer", if no set $B \in \mathcal{F}$ upholds $J(B, Q) \geq j_1$, which is entirely possible depending on the parameters.
Most existing methods to solve this problem depend on the relationship between j_1 and j_2. This relationship is often described as $\rho = \frac{\log_2(1/j_1)}{\log_2(1/j_2)}$. Note that ρ shrinks when j_1 approaches 1.

2.2 Trivial Solution

The most obvious solution is to compare each coordinate in each set $A \in \mathcal{F}$ with each coordinate in Q to calculate the size of the intersection $|A \cap U|$ and union $|A \cup U|$. This solution will always give the correct answer, and

query in $O(d^2n)$ time trivially, but suffers from the *curse of dimensionality.* As the amount of dimensions in the data set doubles, the runtime quadruples! When working with high-dimensional datasets like often seen in text processing, this can have huge consequences. Li, Shrivastava, Moore, *et al.* [5] has shown that it is easy to reach a dimensionality upwards of $d = 2^{83}$ when using *5-shingles* (5 contigous words) of the 10,000 most common English words. A simple improvement is to make a hash table for D of size d with no collisions and look up every entry in A to find any matches. This could reduce the run time to $O(d + dn) = O(dn)$ per query.
These algorithms are both guaranteed to return the set $S = \max_{A \in \mathcal{F}} J(Q, A)$ on every query. The next algorithm can query much faster than this by relaxing this condition: if we allow preprocessing the data and relax the requirement of finding an exact match by considering the *approximate similarity search problem*, we can create sketches of the data set before querying. This sketching strategy can drastically reduce the query time.

2.3 Locality Sensitive Hashing Framework

One of the first big contributions to this problem is the Locality Sensitive Hashing Framework by Indyk and Motwani [2]. This framework aims to solve the Approximate Similarity Search problem for any kind of similarity space, including Jaccard similarity, using some appropriate (s_1, s_2, r_1, r_2)-sensitive family of hash functions that excibit the following properties:

Definition 1. *A family of hash functions $\mathcal{H} = \{h : X \to U\}$ is (s_1, s_2, r_1, r_2)-sensitive for some set X with a similarity function $S : X \to X \to [0; 1]$ if these conditions apply for any $x, y \in X$:*

- *If $S(x, y) \geq s_1$, then $Pr[h(x) = h(y)] \geq r_1$*
- *If $S(x, y) \leq s_2$, then $Pr[h(x) = h(y)] \leq r_2$*

First, we define a family of functions $\mathcal{G} = \{g : X \to U^k\}$ such that $g(p) = \langle h_0(p), \ldots, h_{k-1}(p) \rangle$ for $h_i \in \mathcal{H}, k \in \mathbb{Z}^+$, where k is to be determined later. We can now regard $g(p)$ as the unique identifier of a "bucket", where we can store the value p. We do this l times with l different functions $g_0, \ldots, g_{l-1}$ for each data point $p \in X$. If a collision happens, we store only one of the colliding points at random.
To query, calculate $g_0(q), \ldots, g_{l-1}(q)$ and look up the points in each corresponding bucket. This results in points $P = \{p_0, \ldots, p_{t-1}\}$ where t is the amount of points found. For each $p_i \in P$, if $S(p_i, q) \geq s_2$, then return it. If no points fulfills this constraint, return false.

Part of implementing this framework for a specific similarity metric (e.g. Jaccard Similarity) requires choosing a suitable value for k and l. Finding the right parameters can in this case result in getting a constant 1-sided error probability, as we will show later.

2.4 MinHash

The MinHash algorithm is one of the classic solution to the *approximate similarity search problem*, introduced by Broder [1] for the AltaVista search engine.
It is based on the following theorem:

Theorem 1. *Let g be a hash function from $V \cap W \to [M]$ for some $M \in \mathbb{Z}^+$. Let $H(W) = \min_{w \in W} g(w)$. Then*

$$Pr[H(W) = H(V)] = \frac{|V \cap W|}{|V \cup W|} = J(V, W)$$

Proof. If you sample a random element $x \in V \cup W$, then you can find the probability that $x \in V \cap W$ like so:

$$Pr[x \in V \cap W] = \frac{|V \cap W|}{|V \cup W|} = J(V, W)$$

When we calculate $H(W)$ and $H(V)$, the smallest value of the two will be $H(W \cup V)$. Since we use hash functions, this is essentially equivalent of sampling a random value from $W \cup V$. If this value is in the intersection $W \cap V$, then $H(W)$ and $H(V)$ will have picked the same element. This means that the probability of $H(W) = H(V)$ is the same as the probability of picking an element of the intersection from the union, which is $\frac{|V \cap W|}{|V \cup W|} = J(V, W)$ $\square$

The MinHash algorithm works like so: First, preprocess the data by hashing each coordinate using k different hash functions and saving the smallest value for each hash function as a sketch $S(A) = \langle H_0(A), \ldots, H_{k-1}(A) \rangle$. To query a set Q, calculate its sketch $S(Q)$ and then calculate

$$d(S(A), S(Q)) = \frac{1}{k} \sum_{i \in [k]} [H_i(A) = H_i(Q)]$$

The expected value of this will then be the Jaccard Similarity:

$$\mathbb{E}[d(S(A), S(Q))] = \mathbb{E}[\frac{1}{k} \sum_{i \in [k]} [H_i(A) = H_i(Q)]]$$

$$= \frac{1}{k} \sum_{i \in [k]} \mathbb{E}[[H_i(A) = H_i(Q)]]$$

$$= \frac{1}{k} \sum_{i \in [k]} J(A, Q)$$

$$= J(A, Q)$$

Since each hash function is independent, we can also use a Chernoff probability bound: Let us call each indicator variable $[H_i(A) = H_i(Q)]$ for X_i and let $X = \sum_{i \in [k]} X_i$ and $\mu = \mathbb{E}[X]$. Then for any $\delta > 0$

$$Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right)^{\mu}$$

$$Pr[X > (1-\delta)\mu] < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}} \right)^{\mu}$$

The more hash functions we choose, the larger k and X will be, the larger μ will be and the stricter the bound will be. We can now try to find the run time of the algorithm.
Calculating $H(A)$ of a set A can be done in $O(d)$ time per hash function. Therefore, the preprocessing step can be done in $O(dkn)$ time. To query, one must first calculate the hash of the query set in $|Q|$ time. Then, the query sketch must be compared to each of the sketches in the data set, which can be done in $O(kn)$ time in total. Therefore, it must take $O(|Q| + kn)$ time per query.

2.4.1 Implementation with the LSH framework

For any hash function h picked randomly from $\mathcal{H}$, we know that $Pr[h(A) = h(B)] = J(A, B)$ as shown earlier. This must mean that h is (j_1, j_2, j_1, j_2)-sensitive. If we then consider a function g picked randomly containing $h_0, \ldots, h_{k-1}$ hash functions. We can trivially derive that $Pr[g(a) = g(b)] = J(A, B)^k$. This means that g is (j_1, j_2, j_1^k, j_2^k)-sensitive. By choosing $k = \lceil \frac{\log_2(n)}{\log_2(1/j_2)} \rceil$, we can derive that g now is $(j_1, j_2, O(1/n^{\rho}), 1/n)$-sensitive where $\rho = \frac{\log_2 1/j_1}{\log_2 1/j_2}$. This has the consequence that the error probability now is dependendant on $1/n$, which can be converted to a constant error probability by repeating the experiment multiple times. By keeping $l = \lceil n^{\rho} \rceil$ different data structures, we reduce the error probability to $O(1)$. This means that the run time needed to create the required sketches is $O(l \cdot k \cdot d) = O(n^{\rho} \log_2(n) d)$ per data point. The query time is now $O(l \cdot k \cdot |Q|) = O(n^{\rho} \log_2(n) |Q|)$ since we need to calculate a min-hash value of $|Q|$ for each of the l sketches which takes $O(k \cdot |Q|)$ time per sketch. This process will lead to l different candidate points, from which a true positive can be linearly searched in $O(l \cdot |Q|)$ time.

2.5 Fast Similarity Sketching

Many improvements have since been discovered. The last step of each query of the LSH framework results in an expected $O(L)$ false positives that needs to be filtered out. By using an intermediate sketch of size $O(\log_2^3(n))$, it is possible to reduce the query time to $O((l \cdot |Q|) \log_2^3(n))$ by sampling from the intermediate sketch instead of the regular sketch [6]. Dahlgaard, Knudsen, and Thorup [6] have introduced an improved sketch, which is much faster to create and preserves the same properties as the MinHash sketch, while reducing the query time to $O(k \cdot l + |Q|) = O(n^\rho \log_2(n) + |Q|)$. Part of this improvement is due to a special algorithm that can eliminate bad matches by estimating if the Jaccard similarity between the query set and candidate set is above some threshold without calculating the actual Jaccard similarity. This reduces the filtering step to only take $O(l + |Q|)$ time, since the bad matches can be eliminated in $O(l)$ time and the amount of matches remaining is $O(1)$. Christiani [7] has since improved this query time to $O(n^\rho + |Q|)$.
Until now, the data structures presented error with a constant probability $O(1)$. If one wants to reach a better error probability $1 > \varepsilon > 0$, it is common practice within the field to use $O(\log_2(1/\varepsilon))$ independent data structures. This will usually result in a query time of $O((n^\rho + |Q|) \log_2(1/\varepsilon))$.
Knudsen [10] has shown a method of achieving an improved $O(n^\rho \log_2(1/\varepsilon) + |Q|)$ query time instead, which is very beneficial when working with highly dimensional datasets and a small ε. This is done by creating the M different sketches of Q nescessary for querying from one big sketch of size $M \cdot \log_2^3(n)$, with each sketch segment fed into each LSH datastructure.
Knudsen [10] has further improved the query time to $O((\frac{n \log_2(w)}{w})^\rho \log_2(1/\varepsilon) + |Q|)$ by reducing k such that $h_i \in \mathcal{H}$ becomes $(j_1, j_2, O((n/b)^\rho), n/b)$-sensitive where $b = w/\log_2 w$ and then efficiently filtering through the b expected "bad" matches.
Understanding why this filtering is efficient requires understanding the b-bit minwise hashing trick presented by Li, Shrivastava, Moore, *et al.* [5]: By only storing the b lowest bits of each hash value, one can pack $O(\log_2(w)/b)$ hash values per word (where w is the word size). Knudsen [10] uses 1-bit minwise hashing to create a sketch of size $O(\log_2(w))$ per set $A \in \mathcal{F}$. This allows us to store $O(\frac{w}{\log_2(w)})$ sketches per word.
By calculating the amount of bits per sketch for A and Q that are equal,

one can estimate the Jaccard Similarity to a degree which is good enough to filter a candidate match.
This calculation can be done by bitwise XOR'ing a word packed with sketches of different sets $A \in \mathcal{F}$ with a word packed with sketches of Q. Then, one can calculate the amount of bits set per word. Every bit set will then indicate that Q and A hashed to the same value, which means that we can filter each match based on whether or not the total amount of bits set is over some threshold.
It is this calculation on how to count these bits efficiently that the rest of this project will focus on.

2.6 Parallel Bit Counting

To be able to filter bad matches from good matches, we can use the 1-bit minwise hashing trick to pack the results of multiple subexperiments into one word, which by comparing the bit-wise cardinality (amount of bits set in a bit vector) of two words approximates the similarity between the two. One of the main techniques behind achieving an efficient run time of this method requires computing the cardinality of a bit vector efficiently. Knudsen [10] presents a parallel algorithm to perform this but does neither describe any implementation details and has an unnecessarily complicated proof. I will try to describe the same algorithm in a much simpler fashion and prove its correctness and run time while doing it. A naive algorithm to do this could be described like in algorithm 1.

Algorithm 1 A naive linear time algorithm

```
1: function CARDINALITY($w$, $d$)    ▷ $w$ is the input word, $d$ is the word-size
2:     $x \leftarrow 0$
3:     $i \leftarrow 0$
4:     while $i \leq d$ do
5:         $x \leftarrow x + (w \gg i) \wedge 1$
6:         $i \leftarrow i + 1$
7:     end while
8:     return $x$
9: end function
```

This algorithm trivially runs in linear time to the dimensionality d of the input word w. For n words of size d, this algorithm runs in $O(nd)$ time. Knudsen [10] presents two improvements to this: The first will improve the run time to $O(n \log(d))$ time by utilizing a divide-and-conquer technique and the second to $O(n + \log(d))$ time by introducing parallelism.

2.6.1 Divide-and-Conquer

To explain how divide-and-conquer methods can be used to improve runtime, we must first introduce a bit mask from [10] defined like so:

$$m_{i,j} = \underbrace{0\dots0}_{2^j-2^i}\underbrace{1\dots1}_{2^i}\dots\underbrace{0\dots0}_{2^j-2^i}\underbrace{1\dots1}_{2^i}$$

where $j > i$ and $j, i \in \mathbb{Z}^+$. The notation m_i is a shorthand for $m_{i,i+1}$ and indicates a mask with an equal amount of 1's and 0's. By computing $w \wedge m_{i,j}$ for some word w and some integers i, j, we can isolate a specific segments of size 2^i in the bitstring. We will use this in the following operation:

$$T(w, m, k) = (w \gg k) \wedge m \tag{1}$$

This operation isolates the segments indicated by the bit-mask starting from the kth position of the word w. The algorithm to calculate the cardinality can then be described like in algorithm 2

Algorithm 2 A divide-and-conquer approach

```
1: function CARDINALITY($w$, $d$)    ▷ $w$ is the input word, $d$ is the word-size
2:     $i \leftarrow 0$
3:     while $i \leq \log_2(d)$ do
4:         $w \leftarrow T(w, m_i, 0) + T(w, m_i, 2^i)$
5:         $i \leftarrow i + 1$
6:     end while
7:     return $w$
8: end function
```

The simplest way to gain intuition of this algorithm is to prove it. A loop-invariant is presented like so:

Invariant 1. *At the ith iteration of the algorithm, each bitstring-segment of size 2^i of the word w_i will contain the cardinality of the corresponding segment of the word w_0. Furthermore, the operation can be done in constant time.*

Proof. When $i = 0$, then each bit-string of size $2^0 = 1$ in $w_0 = w$ will be 1 if the bit is 1 and 0 if the bit is 0. This proves our base case.
If at step i the word w_i contains $\frac{d}{2^i}$ segments of size 2^i, then invariant 1 will be upheld if we combine each segment with it's neighbor to form a segment of size 2^{i+1}. Since the cardinality of a combined bitstring is equal to the cardinality of its parts, we can divide all of the segments into pairs of size 2^{i+1} and add them together to form the new pair, which is done by the

operation $T(w, m_i, 0) + T(w, m_i, 2^i)$ in constant time if the masks are precomputed. This operation works because $T(w, m_i, 0)$ isolates every other segment of size 2^i starting from the first segment and $T(w, m_i, 2^i)$ isolates every other segment of size 2^i starting from the second segment.

□

When the algorithm terminates after $\log_2(d)$ iterations, the segments will have size $2^{\log_2(d)} = d$ and thus span the entire original word, which means that we have the bit-count of the original word.

2.6.2 Parallelism

To introduce parallelism into the algorithm, we must first realize the following: When two segments of 2^i bits gets combined, they will not need all of the 2^{i+1} bits to represent their sum. It is actually such that the bits used at iteration i is exactly $i + 1$.

Invariant 2. *At the ith iteration of the algorithm, the amount of bits set in a given segment of a word is at most* $i + 1$*.*

Proof. We will prove this by induction.
At $i = 0$, the size of the segments are $2^0 = 1$, which is equal to $i + 1$.
If it is true at iteration i, then at iteration $i + 1$ the algorithm will add two words of size $i + 1$ which creates a word of size $i + 2$, which fulfills the loop invariant. □

Now, we will introduce a function $l(i)$, which produces the smallest number such that $2^{l(i)} \geq i + 2$. If we use the bit mask $m_{l(i),i+1}$ instead of m_i, we will get the same result. This also means that we can pack $2^{i-l(i)}$ words into one by utilizing the empty space in each segment.

Algorithm 3 A parallel divide-and-conquer algorithm

```
function COMPUTE($S$, $d$)            ▷ $S$ is the input set, $d$ is the word-size
    $n \leftarrow S.length$
    for $i \in [\log_2(d)]$ do
        $k' \leftarrow \{\}$
        $t \leftarrow \{\}$
        for $w \in S$ do
            $k' \leftarrow k' \cup \{T(w, m_i, 0) + T(w, m_i, 2^i)\}$
        end for
        if $i < 2$ then
            $S \leftarrow k'$
        else
            if $l(i) = l(i+1)$ then
                for $j \in [\lceil S.length/2 \rceil]$ do
                    $t \leftarrow t \cup \{k'_{2j} + (k'_{2j+1} \ll 2^i)\}$
                end for
            else
                for $j \in k'$ do
                    $t \leftarrow t \cup \{T(k'_j, m_{l(i)}, 0) + (T(k'_j, m_{l(i)}, 2^{l(i)}) \ll 2^i)\}$
                end for
            end if
            $S \leftarrow t$
        end if
    end for
    $S' \leftarrow \{\}$
    for $j$ in $[S.length]$ do            ▷ For every word in the final set
        for $k$ in $[2^{l(\log_2(d))}]$ do            ▷ For every original word embedded in $S_j$
            $S' \leftarrow S' \cup \{T(S_j, m_{l(\log_2(d)),\log_2(d)+1}, k \cdot 2^{l(\log_2(d))})\}$
        end for
    end for
    return $S'$
end function
```

This algorithm is quite a bit more complex than the first one, and to prove it we need some more terminology.
When we pack the words $t_0, \ldots, t_{2^{i-l(i)}-1}$ into a word t, we say that t is i-packed. This means one can extract all $2^{i-l(i)}$ different words by performing the operation:

$$t_j = T(t, m_{l(i),i+1}, j \cdot 2^{l(i)}) \tag{2}$$

for all $j \in [2^{i-l(i)} - 1]$. The point of the algorithm is to ensure that after every step of the algorithm that every word in S is $(i+1)$ packed. When the first loop of the algorithm terminates at $i = \log_2(d) - 1$, then every word will be $\log_2(d)$-packed, which means that it fits $2^{\log_2(d) - l(\log_2(d))}$ words. When working with 64-bit words, this will result in $2^{6-3} = 8$ words per packed word.

Invariant 3. *At the end of the ith iteration of the first loop of the algorithm, every word in S will be $(i+1)$-packed for any $i \geq 1$*

Proof. First, the first two iterations of the loop are run, such that $i = 1$. During the first two iterations of the loop, no words are combined since it just uses the naive algorithm, which upholds the invariant because any 2-packed words embeds $2^0 = 1$ of the original words.
For any iteration where $i \geq 2$, one out of two things can happen. Either, $l(i+1) = l(i)$ (we do not need more bits to describe each segment), or $l(i+1) = l(i)+1$ (we need twice the amount of bits to describe each segment). In the first case, we will combine the words using bit-shifting. Since every word in k' uses the same amount of bits to represent each segment even though we just combined segments when creating k', there must be an empty space of at least size $l(i+1)$ in every segment. We can fill out that empty space by adding another word bit-shifted by a factor of 2. Now, the amount of words embedded is equal to the sum of both of its parts (i.e. has doubled). This upholds the loop invariant since any $(i+1)$-packed word must have twice the amount of words embedded than a i-packed word if $l(i+1) = l(i)$ by definition.
In the second case, $l(i+1) = l(i) + 1$. In this case, we need to relocate our segments that actually use $2^{l(i)}$ bits so they fit into the segments of size 2^i bits that were created when creating k'. The same amount of words have been embedded into each words as before, but as $l(i+1) = l(i)+1$ a $(i+1)$-packed word should only embed the same amount of words as a i-packed word, which upholds the invariant. Furthermore, since we relocated the segments, we can now still extract each original word by performing the operation in equation 2.
The algorithm will terminate when $i = \log_2(d) - 1$, from which each word will embed $2^{\log_2(d) - l(\log_2(d))}$ of the original words. $\square$

The second for loop of the algorithm extracts the cardinality of each of the original words by using equation 2.
We can now move on to showing the run time of the algorithm. First, we can try to find the run time of the first for-loop. At each iteration i, we will describe the amount of elements in S as n_i with $n = n_0$. We can then

describe n_i like so:

$$m_i = \lceil \frac{m}{2^{i-l(i)}} \rceil \tag{3}$$

which means that the first for-loop in total takes

$$\sum_{i=2}^{\log_2(d)} \lceil \frac{m}{2^{i-l(i)}} \rceil$$

Since the last loop iterates over every element of the original set, it must take $O(m)$ time. We can now derive the total run time:

$$O(m) + \sum_{i=2}^{\log_2(d)} \lceil \frac{m}{2^{i-l(i)}} \rceil$$

$$= O(m) + \sum_{i=2}^{\log_2(d)} \lceil m2^{l(i)-i} \rceil$$

$$\leq O(m) + \sum_{i=2}^{\log_2(d)} (m2^{l(i)-i} + 1)$$

$$\leq O(m + \log(d)) + m \sum_{i=2}^{\log_2(d)} 2^{l(i)-i}$$

Now we use the fact that $l(i)$ is always the smallest number that satisfies $2^{l(i)} \geq i + 2$. Since it is the smallest number, then $2^{l(i)} < 2(i + 2)$, which can be proved by contradiction: $2^{l(i)} \geq 2(i+2) \implies 2^{l(i)-1} \geq i + 2$, which shows that there is a number that is 1 smaller than $l(i)$ that upholds the bound $2^{l(i)} \geq i + 2$. Therefore $2^{l(i)} < 2(i + 2)$. We can now use this fact:

$$\leq O(m + \log(d)) + m \sum_{i=0}^{\infty} \frac{2(i+2)}{2^i} = O(m + \log(d))$$

The run time of the algorithm can therefore be bound to $O(m + \log(d))$.

3 Methods

3.1 Hypothesis

We are interested in quantifying whether the parallel bit-counting algorithm presented by Knudsen [10] is faster than the trivial linear time model. To do this, we need to formulate a null-hypothesis:

Hypothesis 1. *Parallel Bit Counting as described by Knudsen [10] is not faster than a simple linear-time algorithm in practice.*

To reject this hypothesis, we are interested in measuring the actual run time on some kind of data and compare the results to some basis model.

3.2 Benchmarking

To be able to quantify whether one algorithm is faster than the other, one can try to implement it in practice, run it on a specified data set and measure how long it takes for the algorithm to terminate. Benchmarking this way is common in the industry, but it can be hard to be able to generalise from results, due to the many possible sources of error such as:

- Results can vary from machine to machine due to factors like difference in instruction sets, processor infrastructure, cache hierachy, compiler compatibility, memory capacity etc.
- The input dataset can be biased towards one specific kind of architecture or implementation.
- Caches become more efficient when "warmed up" e.g. when the computer has recently fetched from the same memory addresses. Two identical benchmarks might produce different results based on whether or not the caches are filled.
- A process that benchmarks a program will have to share the CPU cores with other processes running on the computer. This can produce a high variance between identical benchmarks on the same machine.
- Different compilers can produce more og less optimized code, which might produce varying results on different compilation targets.

The scope of this project disallows running benchmarks on a wide varity of computers, which presents some uncertancy in the validity of the results. Some of the other sources of error are mitigable.

- The benchmark should be run multiple times with different pseudo-randomly generated input to mitigate bias in the input data.
- The benchmark should be run multiple times per input data set, until a significant mean runtime can be calculated.
- The benchmark should perform a few ”dry runs” before each run to warm up the caches.

This applications of this software lends itself to large-scale machine learning applications, search engines and data mining. Therefore, the most relevant results would come from running the benchmarks on a machine with access to a GPU, and for the algorithm to be written with parallelism[1] in mind. This has not been considered within the scope of this project however, but instead lends itself as an open research problem.

[1]In this case, parallelism is meant in a GPU context unlike word parallelism as mentioned previously

4 Implementation

4.1 Technology

The implementation to be measured upon has been written in Rust, a memory safe, fast, compiled programming language[12] that has great tooling and typing for writing, testing and benchmarking algorithms.
It uses the `rustc` compiler with a `LLVM` compilation back-end. It does not use a garbage collector but relies on compile time borrow checking, which leads to a language that is comparable to or faster than C[9].
The binary to be benchmarked is compiled and run on a laptop workstation as described in appendix A.

4.2 Design

To be able to use the bit mask, it needs to be calculated efficiently. A $\mathbb{Z}^{\log_2(d)\times\log_2(d)}$ matrix can be calculated in $O(\log_2(d)^2)$ time at the start of the procedure. Then, each bit mask can then be looked up in constant time.
It is also nescessary to efficiently be able to look up $l(i)$ for any iteration i. Since the word size is known on compile time and since at most $\log_2(d)+2$ different values are needed, each possible value can easily be hard-coded as a pattern match statement.

4.3 Correctness

It is luckily easy to verify the correctness of the implementation of the algorithm since it is a deterministic algorithm. That means that it is possible to calculate any expected output by hand, which both is useful for validating and for debugging.
Which inputs should be used though? To be able to show that the implementation works on a wide array of inputs, we can generate a large list of random numbers to help cover eventual edge cases. We can then compare the results to some reference implementation - in this case the built-in `count_ones()` method.
It is much harder to verify whether the code works with multiple different word-sizes. This is due to the fact that the else branch on line 16 of algorithm 3 only gets hit once when working with $d = 64$ (when $i = 2$) and twice when $d = 128$. Since the applications of this algorithm really benefits when using large word sizes, it is very relevant that this gets tested. A word size of $d = 256$ was used as well to show that the else-statement path as described earlier also works as intended.

4.4 Benchmarking

To benchmark the algorithm, two reference implementations were created to be used as a baseline models. These implementations were based on algorithm 1 and 2. A list of 2^{24} random numbers were to be generated and used as input for each algorithm.
The Rust library `Criterion.rs`[11] was used to perform the benchmarking. Criterion.rs was very useful for multiple reasons.

- First of all, Criterion.rs integrates into standard Rust tooling easily and allows benchmarking on binary code built for production (e.g. with full optimizations and no debugging symbols)
- Secondly, Criterion.rs provides a compiler "opaque" functions that makes sure that repeated calls to the same function do not get optimized away.
- Criterion.rs handles warming up caches by running the algorithms multiple times before beginning to measure.
- Lastly, Criterion.rs does not record the run time of a single operation - instead it runs multiple samples each consisting of a number of iterations. The run time is the calculated as the runtime per sample divided by the amount of iterations.

5 Results

6 Discussion

7 Conclusion

A Machine Specifications

Model	Lenovo Thinkpad E580
Operating System	Arch Linux kernel 5.15.5-arch1-1
Word size	64bit
Processor model	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
ISA	x86_64
Memory Capacity	32 GB
Caches	128 KiB L1d, 128 KiB L1i, 1MiB L2, 8 MiB L3

References

[1] A. Broder, *On the resemblance and containment of documents*, 1997. DOI: `10.1109/SEQUEN.1997.666900`.

[2] P. Indyk and R. Motwani, “Approximate nearest neighbors: Towards removing the curse of dimensionality,” in *STOC ’98*, 1998.

[3] A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimensions via hashing,” *Proceeding VLDB ’99 Proceedings of the 25th International Conference on Very Large Data Bases*, vol. 99, May 2000.

[4] P. Li and A. C. König, “Theory and applications of b-bit minwise hashing,” *Commun. ACM*, vol. 54, no. 8, pp. 101–109, Aug. 2011, ISSN: 0001-0782. DOI: `10.1145/1978542.1978566`. [Online]. Available: `https://doi.org/10.1145/1978542.1978566`.

[5] P. Li, A. Shrivastava, J. Moore, and A. C. Konig, *Hashing algorithms for large-scale learning*, 2011. arXiv: `1106.0967 [stat.ML]`.

[6] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, *Fast similarity sketching*, 2017. arXiv: `1704.04370 [cs.DS]`.

[7] T. Christiani, *Fast locality-sensitive hashing frameworks for approximate near neighbor search*, 2018. arXiv: `1708.07586 [cs.DS]`.

[8] S. Vassilvitskii, *Coms 6998-12: Dealing with massive data*, 2018.

[9] P. Emmerich, S. Ellmann, F. Bonk, A. Egger, E. G. Sánchez-Torija, T. Günzel, S. D. Luzio, A. Obada, M. Stadlmeier, S. Voit, and G. Carle, “The Case for Writing Network Drivers in High-Level Programming Languages,” in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019), ¡strong¿Best Paper Award¡/strong¿*, Sep. 2019.

[10] J. B. T. Knudsen, *Fast similarity search with low error probability*, 2021.

[11] *Criterion.rs*, `https://github.com/bheisler/criterion.rs`, Accessed: 2021-12-09.

[12] *Rust programming language*, `https://www.rust-lang.org/`, Accessed: 2021-12-06.