UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor's project

Parallel Bit-Counting

for Approximate Similarity Searching

Rasmus Hag Løvstad, pgq596@alumni.ku.dk

Supervisor: Mikkel Thorup

Handed in: January 15, 2022

Abstract

This project regards an analysis of recent advances in solving the *Approximate Similarity Search Problem* with respect to the *Jaccard Similarity* $J(A, B) = \frac{A \cap B}{A \cup B}$, for some subsets A, B of some universe U. Given a corpus $\mathcal{F}$ consisting of n subsets of U, two constants $0 < j_2 < j_1 < 1$ as well as a set $Q \subseteq U$ find a set $A \in \mathcal{F}$ such that $J(A, Q) \geq j_2$ if there exists a set $B \in \mathcal{F}$ where $J(B, Q) \geq j_1$.
First, existing solutions to this problem will be described, such as the *Locality Sensitive Hashing Framework* (LSH) by Gionis, Indyk, and Motwani [3], the seminal *MinHash* algorithm by Broder [1] and the *Fast Similarity Sketching* algorithm by Dahlgaard, Knudsen, and Thorup [7]. Then the focus will be shifted to the most recent advances by Knudsen [11], who presents a parallel bit-counting algorithm that supposedly can make the query time sublinear to the amount of sets in $\mathcal{F}$.
This project investigates the efficiency of counting bits in parallel by evaluating the algorithm in terms of time complexity, proof of correctness and an empirical comparison to simpler methods. The findings include a theoretical time advantage to using parallel bit counting compared to a simple, linear time algorithm, but empirical results show that this advantage is hard to achieve in practice. Furthermore, reflections are made on how these results might scale on more specialized hardware, wherein the Approximate Similarity Search Problem may find the most use.

Contents

1 Introduction

The *Approximate Similarity Search Problem* regards efficiently finding a set $A \in \mathcal{F}$ from some corpus $\mathcal{F}$ consisting of n subsets of some universe U that is approximately similar to a query set Q in regards to the *Jaccard Similarity* metric $J(A,Q) = \frac{|A \cap Q|}{|A \cup Q|}$[7][11]. Approximate means that given two variables $0 < j_2 < j_1 < 1$, the algorithm should 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 $J(B,Q) \geq j_1$. Practical applications includes searching through large corpi of high-dimensional text documents like plagiarism-detection or website duplication checking among others[9]. The main bottleneck in this problem is the *curse of dimensionality*. A trivial algorithm can solve this problem in $O(n|Q| \max_{A \in \mathcal{F}} |A|)$ time (where n is the amount of elements in the corpus $\mathcal{F}$), but algorithms with a query-time proportional to the dimensionality of the corpus $\max_{A \in \mathcal{F}} |A|$ 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.* [6] shows easily can reach a dimensionality upwards of $|Q| = 2^{83}$ 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 $O(n^\rho \log(n)|Q|)$ time (where $\rho = \frac{\log_2(1/j_1)}{\log_2(1/j_2)}$), a significant improvement independent of the dimensionality of the corpus. Many improvements have since been presented to both improve preprocessing time, query time and space efficiency. Notable mentions includes (but are not limited to) the use of *tensoring*[4], *b-bit minwise hashing*[5], *fast similarity sketching*[7]. Applications of these techniques lead to efficient querying in $O(n^\rho + |Q|)$ time, with a constant error probability. If one wishes to achieve an even better error probability such as $\varepsilon = o(1)$, it is standard practice within the field to use $O(\log_2(1/\varepsilon))$ independent data structures and return the best result, resulting in a query time of $O(\log_2(1/\varepsilon)(n^\rho + |Q|))$.

Recent advances by Knudsen [11] show that it is possible to achieve an even better query time by sampling these data structures from one large sketch, leading to a query time of $O(n^\rho \log_{1/\varepsilon} + |Q|)$.

Each of the data structures will return a list of candidate sets, of which false positives needs to be eliminated from. An efficient filtering scheme can choose exactly 1 set from each data structure as a candidate. Then, the final result can be picked by eliminating data structures that are highly probable to return a bad candidate. This final step requires computing the cardinality of a list of bit-strings (e.g. counting the amount of bits set) very efficiently, for which an algorithm is presented by Knudsen [11] that brings the query time down to $O((\frac{n \log_2 w}{w})^\rho \log(1/\varepsilon) + |Q|)$ where w is the word-size.

The main focus of this project is to analyse, prove, implement and evaluate this parallel bit-counting technique. The analysis will be based on the original paper [11], but with some modifications to resolve some of the issues with the original algorithm. This will also include a pseudo-code implementation of the algorithm since the original paper only describes it through recurrences. This leads to a proof of correctness that slightly differs from the one presented in the paper, and a time complexity analysis that does indeed show the sublinear running time as claimed.

The theoretical analysis will be backed up by a real-life implementation that can be

benchmarked to help show this sublinear running time in practice. At last, reflections on the results and methods will be made to back up eventual conclusions.

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. It is furthermore only defined when both A and B are non-empty, as to avoid dividing by zero.
This part aims to define the problem to be considered in a concise way. Then, existing solutions will be described to set the context for the main focus of this project: A parallel bit-counting algorithm to be used for filtering potential matches for the similarity search algorithms.

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 $Q \subset U$, preprocess $\mathcal{F}$ to efficiently find the set $A \in \mathcal{F}$ such that $A = \max_{A \in \mathcal{F}} J(A, Q)$.
In many practical applications, it may be enough to only consider the *Approximate Jaccard Similarity Search Problem* (as defined by Knudsen [11]): Given the setting above and two constants $0 < j_2 < j_1 < 1$, 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" (if a good solution exists). 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 as j_1 approaches 1. The query time of most solutions use ρ as a exponent of n, which means that the lower the value of ρ is, the fewer elements in the corpus needs to be checked.

2.2 Trivial Solution

The most obvious solution is to compare each element in each set $A \in \mathcal{F}$ with each element 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 $O(n|Q| \max_{A \in \mathcal{F}} |A|)$ time trivially, but suffers from the *curse of dimensionality*. As the amount of dimensions in the data set doubles, the running time quadruples! When working with high-dimensional datasets like often seen in text processing, this can have huge consequences. Li, Shrivastava, Moore, *et al.* [6] has shown that it is easy to reach a dimensionality upwards of $|A| = 2^{83}$ 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 Q of size $\max_{A \in \mathcal{F}} |A|$ with no collisions and look up every entry in A to find any matches. This could reduce the running time to $O(|Q| + n \max_{A \in \mathcal{F}} |A|)$ per query.
These algorithms are both guaranteed to return the set $S = \arg\max_{A \in \mathcal{F}} J(Q, A)$ 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 instead, we can create so-called 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.

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

- *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 $\mathcal{G} = \{g \colon X \to U^k\}$ such that for each $g \in \mathcal{G}$, $g(p) = \langle h_0(p), \ldots, h_{k-1}(p) \rangle$ for $h_i \in \mathcal{H}$, where $k \in \mathbb{Z}^+$ is to be determined later. We can now regard $g(p)$ as the index of a "bucket", where we can store the value p. We do this l times with the functions $g_0, \ldots, g_{l-1} \in \mathcal{G}$ for each data point $p \in X$. If a collision happens, we store only one of the colliding points, chosen at random.
To query, look up the corresponding point for each of the buckets $g_0(q), \ldots, g_{l-1}(q)$. This yields the points $P = \{p_0, \ldots, p_{t-1}\}$ where $t \leq l$ is the amount of points found. For each $p_i \in P$, if $S(p_i, q) \geq j_2$, 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 error probability, as we will show later.

2.4 MinHash

The MinHash algorithm is one of the classic solutions 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 h be a universal hash function from $V \cup W \to [M]$ for some $M \in \mathbb{Z}^+$ where $W \cup V \subseteq U$ for some universe U. Let $H(W) = \min_{w \in W} h(w)$. 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 $W \cup V$. Assuming no collisions, this value will be in the intersection $W \cap V$ if and only if $H(W)$ and $H(V)$ 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 $\frac{|V \cap W|}{|V \cup W|} = J(V, W)$. □

The MinHash algorithm works like so: First, preprocess each set $A \in \mathcal{F}$ by hashing each coordinate using k universal hash functions picked uniformly at random, 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

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

We can then derive the expected value of this to be the Jaccard Similarity:

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

Using linearity of expectation:

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

Using theorem 1:

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

2.4.1 Implementation with the LSH framework

For a function $H \in \mathcal{H}$ from some family $\mathcal{H}$ defined as $H(W) = \min_{w \in W} h(w)$ as in theorem 1, we know that $Pr[H(A) = H(B)] = J(A, B)$ as per theorem 1. This must mean that $\mathcal{H}$ is (j_1, j_2, j_1, j_2)-sensitive for some values $0 < j_2 < j_1 < 1$ as per definition 1 and theorem 1:

- If $J(W, V) \geq j_1$, then $Pr[H(W) = H(V)] \geq j_1$
- If $J(W, V) \leq j_2$, then $Pr[H(W) = H(V)] \leq j_2$.

We can then consider a function $g \in \mathcal{G}$ from some family $\mathcal{G}$ defined as $g(W) = \langle H_0(W), \ldots, H_{k-1}(W) \rangle$ where functions $H_0 \ldots H_{k-1}$ are picked uniformly at random

from $\mathcal{H}$. An evaluation of any $g \in \mathcal{G}$ will be called a "sketch" from now on. Since the functions $H_0 \dots H_{k-1}$ are independent, we can use Bayes' theorem to derive that:

$$Pr[g(W) = g(V)] = Pr[\bigcap_{H \in g} H(W) = H(V)]$$

$$= \prod_{H \in g} Pr[H(W) = H(V)]$$

$$= \prod_{H \in g} J(W,V) = J(W,V)^k$$

This means that $\mathcal{G}$ is (j_1, j_2, j_1^k, j_2^k)-sensitive for some $k \in \mathbb{Z}^+$ since $J(W,V)$ and j_1, j_2 are non-negative:

- If $J(W,V) \geq j_1$, then $J(W,V)^k \geq j_1^k$, which means that $Pr[g(W) = g(V)] \geq j_1^k$
- If $J(W,V) \leq j_2$, then $J(W,V)^k \leq j_2^k$, which means that $Pr[g(W) = g(V)] \leq j_2^k$

By choosing $k = \lceil \frac{\log_2(n)}{\log_2(1/j_2)} \rceil$, we can derive that $\mathcal{G}$ now is $(j_1, j_2, 1/n^\rho, 1/n)$-sensitive where $\rho = \frac{\log_2 1/j_1}{\log_2 1/j_2}$.

$$j_1^k = j_1^{\lceil \frac{\log_2(n)}{\log_2(1/j_2)} \rceil} \geq j_1^{\frac{\log_2(n)}{\log_2(1/j_2)}}$$

$$= j_1^{\frac{\log_2(n)}{\log_2(1/j_2)} \cdot \frac{\log_2(1/j_1)}{\log_2(1/j_1)}}$$

$$= j_1^{\frac{\log_2(1/j_1)}{\log_2(1/j_2)} \cdot -\frac{\log_2(n)}{\log_2(j_1)}}$$

$$= j_1^{-\rho \frac{\log_2(n)}{\log_2(j_1)}}$$

$$= j_1^{-\rho \log_{j_1}(n)}$$

$$= \frac{1}{n^\rho}$$

$$j_2^k \leq j_2 \cdot j_2^{\frac{\log_2(n)}{\log_2(1/j_2)}} = j_2 \cdot j_2^{-\frac{\log_2(n)}{\log_2(j_2)}} = \frac{j_2}{n} \leq 1/n$$

Now, imagine $g_0, \dots g_{l-1} \in \mathcal{G}$ picked uniformly at random where $l = \lceil n^\rho \rceil$. If we calculate the sketch $g_i(A)$ for each $A \in \mathcal{F}$, $i \in [l]$, we will have $O(n \cdot l)$ sketches where $n = |F|$. This is our LSH data structure.

For each value $i \in [l]$, a hash table should also be created to look up a set stored at index $g_i(Q)$ for some query set Q in constant time. Furthermore, each set $A \in \mathcal{F}$ stored in this hashtable should have an associated hash table over its elements.

Creating the data structure takes $O(\sum_{A \in \mathcal{F}} |A| \cdot l)$ time and space during the preprocessing step.

To query, we first calculate l sketches of Q, $g_0(Q), \dots, g_{l-1}(Q)$, which takes $O(k \cdot l \cdot |Q|)$ time. These sketches can be used to retrieve l candidate sets $\mathcal{M} = \{A_0, \dots A_{l-1}\}$ from each of the l outermost hash tables.

For each $A \in \mathcal{M}$, the Jaccard similarity can be computed in $O(|Q|)$ time by computing the size of the intersection using the associated hash table to A and computing the size of the union by the inclusion-exclusion principle if the size of A is known.

When an element $A \in \mathcal{M}$ is found such that $J(A,Q) \geq j_2$, the algorithm can return.

This algorithm can therefore query in $O(l \cdot k|Q| = \mathrm{O}(\mathrm{n}^\rho \log_2(n)|Q|)$ time.

2.5 Later Improvements

Many improvements have since been discovered. Andoni and Indyk [4] have since improved the total query time to $O(n^\rho|Q|)$ by using a technique called *tensoring*. Tensoring entails creating a collection of $m = o(l)$ sketch functions instead of the usual k by picking m functions from $\mathcal{H}$ uniformly at random. Let each $g_i$for $i \in [l]$ from the LSH framework now be a distinct combination of these m hash functions of size t (where t is a positive integer). It is now possible to choose t such that $\binom{m}{t} = l$, such that every combination of the m hash values are used. By doing this, it is only nescessary to compute $O(l)$ hash values of Q which is a reduction by a factor of k [4][8].

Dahlgaard, Knudsen, and Thorup [7] have also introduced an improved sketch called the *Fast Similarity Sketch* (FSS), which is much faster to create and preserves the same properties as the MinHash sketch, while reducing the query time to $O(k \cdot l + |Q|) = O(n^\rho \log_2(n) + |Q|)$.

This improvement in query time is due to multiple factors.

First of all, a sketch of size k of a set A takes an expected running time of $O(k \log_2(k) + |A|)$ to create using the FSS algorithm, which is improved compared to the $O(k|A|)$ running time of the classic MinHash algorithm.

Secondly, the algorithm uses a single intermediate sketch that it samples the hash values of the sketch from, where the size of the intermediate sketch is independent of the size of the query set, which turns the time to create the sketch of Q from $O(n^\rho \log_2(n)|Q|)$ to $O(n^\rho \log_2(n) + |Q|)$.

Thirdly, the algorithm also uses a faster method for eliminating "bad matches" from the $O(l)$ candidate sets by estimating if the Jaccard similarity between the query set and candidate set is above some threshold within a certain probability. Here, a "bad match" is defined as a candidate set A where $J(A, Q) < j_2$. 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 $O(1)$. These $O(1)$ matches can then finally be compared with to the query set which takes $O(|Q|)$ time.

These three factors result in a $O(n^\rho \log_2(n) + |Q|)$ total query time, which can be combined with tensoring to a $O(n^\rho + |Q|)$ query time as shown by Christiani [8].

Until now, the data structures presented error with a constant probability $O(1)$. If one wants to reach a better error probability ε (could for example be $\varepsilon = O(1/n)$), it is common practice within the field to use $M = \Theta(\log_2(1/\varepsilon))$ independent data structures. This will usually result in a query time of $O((n^\rho + |Q|) \log_2(1/\varepsilon))$.

Knudsen [11] 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 sampling the $\Theta(M \cdot l)$ different sketches of Q nescessary for querying from a single sketch of size $\Theta(M \cdot \log_2^2(n))$, with each sketch segment fed into each LSH datastructure.

Knudsen [11] has further improved the query time to $O((\frac{n \log_2(d)}{d})^\rho \log_2(1/\varepsilon) + |Q|)$ (where d is the word-size) by reducing k such that $h_i \in \mathcal{H}$ becomes $(j_1, j_2, O((n/b)^\rho), n/b)$-sensitive where $b = d/\log_2(d)$ 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.* [6]: By only storing the b lowest bits of each hash value, one can greatly reduce the storage space required for minwise hashing. Knudsen [11] uses 1-bit minwise hashing to create a sketch of

$O(\log_2(d))$ hash values per set $A \in \mathcal{F}$. While 1-bit hashing greatly increases the chance of collision per hash value, it allows us to use many more hash values per sketch, which is suprisingly effective at negating this [6]. This allows us to store $O(\frac{d}{\log_2(d)})$ sketches per word. When querying, instead of comparing our sketches of Q with each of the l candidate sketches one at a time, we can compare it to $O(\frac{d}{\log_2(d)})$ different sketches at once.

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 XNOR'ing a word packed with sketches of a set $A \in \mathcal{F}$ with a word packed with sketches of Q. Then, one can calculate the amount of bits set. 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.

3 Parallel Bit Counting

In the same article where Knudsen [11] presents the improvements for algorithms with an $o(\varepsilon)$ error probability, a parallel algorithm to compute the cardinality of a bit-string is also presented. It is only described by recurrences, and omits implementation details. I will try to describe the same algorithm in a much simpler fashion and prove its correctness and time complexity while doing it. This means that this section is my own interpretation of the algorithm, proofs and analysis described in [11]. A naive algorithm to perform bit-counting 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 ← 0
3:     for i ∈ [d] do
4:         x ← x + (w ≫ i) ∧ 1
5:     end for
6:     return x
7: end function
```

This algorithm trivially runs in linear time to the dimensionality[1] d of the input word w. For n words of size d, this algorithm runs in $O(nd)$ time. Knudsen [11] presents two improvements to this: The first will improve the running time to $O(n \log(d))$ by utilizing a divide-and-conquer technique and the second to $O(n + \log(d))$ time by introducing parallelism.

3.1 Divide-and-Conquer

To explain how divide-and-conquer methods can be used to improve running time, we must first introduce a word-sized bit mask from [11] that is defined like so:

$$m_{i,j} = \underbrace{0 \ldots 0}_{2^j - 2^i}\underbrace{1 \ldots 1}_{2^i} \ldots \underbrace{0 \ldots 0}_{2^j - 2^i}\underbrace{1 \ldots 1_2}_{2^i}$$

where $j > i$ and j and i are non-negative integers. 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 replace every other sub-string of size $2^j - 2^i$ in the bit-string with 0's. This will be described as *isolating* segments of size 2^i.
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. This is useful for partitioning a bit-string, since any word now can be described as

$$w = T(w, m_i, 0) + (T(w, m_i, 2^i) \ll 2^i)$$

[1] For the rest of this project, it is assumed that d is a power of 2.

Notice that even though we use addition, overflows are impossible since the segment pattern of $T(w, m_i, 0)$ and $(T(w, m_i, 2^i) \ll 2^i)$ do not overlap.
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:     for i ∈ [log_2(d)] do
3:         w ← T(w, m_i, 0) + T(w, m_i, 2^i)
4:     end for
5:     return w
6: 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 beginning of the ith iteration of the algorithm, the word $w^{(i)}$ can be regarded as consisting of $d/2^i$ bit-string-segments of size 2^i, that represents the cardinality of the corresponding segments of the word $w^{(0)}$.*

Proof. **Base case**: 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.
Induction step: At step i, invariant 1 states that the word $w^{(i)}$ contains $\frac{d}{2^i}$ segments of size 2^i. The operation $T(w, m_i, 0)$ will then isolate exactly every other segment of size 2^i. The operation $T(w, m_i, 2^i)$ will isolate the remaining segments and bit-shift them such that the segments align with the segments isolated by $T(w, m_i, 0)$.
By adding $T(w, m_i, 0)$ and $T(w, m_i, 2^i)$ together, every pair of overlapping segments will be added together to form a new segment of double the size. Invariant 1 states that each segment of $T(w, m_i, 0)$ and $T(w, m_i, 2^i) \ll 2^i$ represents the cardinality of a corresponding segment in $w^{(0)}$. Since the cardinality of a bit-string is equal to the sum of its parts, we can interpret each new segment as the cardinality of the corresponding two previous segments combined. Therefore, at the end of the i-th iteration of the for-loop, the word w can be interpreted as consisting of bit-string segments of size 2^{i+1}, where each bit-string-segment represents the cardinality of the corresponding bit-string segments of $w^{(0)}$. □

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. If m is pre-computed, then each iteration can be performed in constant time, which means that the cardinality of a list of n words can be calculated in $O(n \log_2(d))$ time.

3.1.1 Example

Given the bit-string $w^{(0)} = 1101011001010010_2$, we can run the algorithm by hand to find the cardinality. Due to invariant 1, we can regard $w^{(0)}$ as consisting of $d/2^0 = 16$ sections of size $2^0 = 1$ as marked by colors:

$$w^{(0)} = 1101011001010010_2$$

At the beginning of the first iteration, we can calculate m_0 to be $\ldots 0101010101010101_2$. This means that $T(w^{(0)}, m_0, 0) = 0101010001010000_2$ and $T(w^{(0)}, m_0, 2^0) = 0100000100000001_2$. We can add them together to get $w^{(1)}$:

$$w^{(1)} = 1001010101010001_2$$

Notice that each section has now doubled in size. Due to invariant 1, each section represents the bit-string cardinality of the corresponding section of $w^{(0)}$. For example, the least significant section $01_2 = 1_{10}$ represents that the there is 1 bit set in the 2 least significant bits of $w^{(0)}$ (which we can see is true). Continuing the algorithm, we get:

$$w^{(2)} = 0011001000100001_2$$

$$w^{(3)} = 0000010100000011_2$$

$$w^{(4)} = 0000000000001000_2 = 8_{10}$$

The algorithm has now terminated, and we can see that $w^{(0)}$ contains 8 bits set, which we can see is true.

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

Proof. **Base case**: At $i = 0$, the size of the segments are $2^0 = 1$, which is equal to $i + 1$.
Induction step: At iteration i, the algorithm will add two words of size 2^i. In the worst case, where all bits are set, each segment in $w^{(i)}$ will have the value $2^{i+1}-1$, which means that each segment in $w^{(i+1)}$ will have the value $2^{i+1}-1+2^{i+1}-1 = 2(2^{(i+1)}-1)$. To find out how many bits is needed to represent the given number, we can calculate $\log_2$ of the expression and round down: $\lfloor \log_2 (2(2^{(i+1)} - 1)) \rfloor = \lfloor 1+\log_2 (2^{(i+1)} - 1) \rfloor = 1 + \lfloor \log_2 (2^{(i+1)} - 1) \rfloor = i + 1$. Therefore, at the end of the ith iteration, each section can only have $i + 1$ or fewer bits set. □

Now, we will introduce a function $l(i)$, which produces the smallest non-negative integer such that $2^{l(i)} \geq i + 2$. We can now use the bit-mask $m_{l(i),i+1}$ instead of m_i, since the segments are guaranteed to be strictly smaller than $2^{l(i)}$. At the end of each iteration, there will be a sequence of $2^{(i+1)-l(i+1)}$ unused bits between each segment, since each segment will have size $2^{(i+1)}$ (invariant 1) but at most use $i+1$ bits (invariant 2). In the case that $2^{(i+1)-l(i+1)} \geq 2^{l(i+1)}$, this unused sequence can contain a segment from an entirely different word (note that this happens when $l(i) = l(i+1)$). Algorithm 3 uses this to efficiently compute the cardinality of all the words in a set $S = \{w_0, \ldots w_{n-1}\}$.

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 $l(i) = l(i+1)$ AND $S.length > 1$ **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. □

3.2.1 Example

For the first two iterations of the first for-loop, the algorithm works exactly as algorithm 2. When $i = 2$, lines $17 - 19$ are hit for the first time, but since the words are (3)-packed, there will only be $2^{2-2} = 1$ words from S^0 embedded in each word in $S^{(2)}$. This results in the operation at line 18 leaving k' unchanged.
When $i = 3$, the lines $12 - 16$ are hit for the first time. This results in words getting pairwise combined. Here is an example of how words in k' for some set $S^{(3)}$ can get merged together (segments of size $2^{l(i)}$ are colored):

$$k' =$$

$$\{00000000\textcolor{red}{00001111}00000000\textcolor{red}{00001001}_2, 00000000\textcolor{blue}{00001010}00000000\textcolor{blue}{00010000}_2\}$$

$$S^{(4)} = \{\textcolor{blue}{00001010}\textcolor{red}{00001111}\textcolor{blue}{00010000}\textcolor{red}{00001001}_2\}$$

Since lines 6-8 of the next iteration of the algorithm will combine segments of size $l(4) = l(3)$, k' will be:

$$k' = \{0000000000000000\textcolor{blue}{00011010}\textcolor{red}{00011000}_2\}$$

After this iteration, the main for-loop terminates and the final set S' can be constructed to the following:

$$S' = \{00000000000000000000000\textcolor{red}{00011000}_2, 00000000000000000000000\textcolor{blue}{00011010}_2\}$$

$$= \{24_{10}, 26_{10}\}$$

This means that $S_0^{(0)}$ should contain 24 bits set and $S_1^{(0)}$ should contain 26 bits set.

3.3 Discrepansies

In the original article, combining the sections of a word (like we do in algorithm 2 and on line 7 and 14 in algorithm 3) is done with the following operation:

$$t^{(i+1)} = T(T(t^{(i)}, m_i, 0) + T(t^{(i)}, m_i, 2^i), m_{i+1}, 0)$$

Here, the algorithm is described as a recurrence, where $t^{(i)}$ is the word t after i iterations. This definition is troublesome since the outermost call to T causes information loss. A counter-example to this is just the word $t^{(0)} = 1111_2$. We can see that this word contains 4 bits set. Using this operation, we can see that

$$t^{(1)} = T(T(1111_2, 0101_2, 0) + T(1111_2, 0101_2, 2), 0011_2, 0) = 0010_2$$

The outermost application of the bit mask means that we lose information about the two most significant bits of t. If we were to calculate $t^{(2)}$ and thus let the algorithm terminate, we would only have counted half of all the bits!
This is simply solved by removing the outermost call to T, since no information will be lost and invariant 1 will apply.
Another issue with the original article is that the order of the cardinalities in the final set S' is not consistent with the order of the original set $S^{(0)}$. When the algorithm shifts the sections in each word in line 17-19 of the algorithm, every other section gets shifted 2^i bits to the left. When the algorithm has terminated, the cardinality of $S_i^{(0)}$ might be contained in S'_j where $i \neq j$. The original paper by Knudsen [11] does not mention this, and it is not said whether this affects the criteria of correctness, but for any practical usage, this does seem like it could be an issue. This is only a problem when $\log_2(d) \leq 7$, since the affected branch is only run when $l(i+1) = l(i) + 1$ and $i > 2$ (it is also run when $i = 2$, but causes no problem since no words have been packed yet, which means that they will not get shifted). In the case where $d = 128$, the following formula calculates an index j from i such that $|S_i| = S'_j$.

$$j(i) = \begin{cases} (i \gg 1) + ((i \gg 3) \ll 2) & \text{if } i\&1 = 0 \\ (i \gg 1) + 4 + ((i \gg 3) \ll 2) & \text{otherwise} \end{cases}$$

Proof. When $i = 6$, every word in S contains $2^{6-3} = 8$ words. Before lines 17-19 are run, each word from the original set is described as one bit-string segment of $2^{l(6)} = 8$ bits each. Lines 17-19 then takes every other segment and shifts it $2^6 = 64$ bits to the left. Since this is the last iteration of the algorithm, the for loop in lines 25-29 gets run immediately after, which extracts each segment into S' in the order of the least significant first for each word.
This means that every other word from the original set will be shifted 4 indices later in S', since 64 bits can contain $64/8 = 8$ words and half of them get shifted as well. The calculation presented to find $j(i)$ has two branches: one branch if i is even and one if i is uneven. If i is even, $j(i)$ can be found by calculating $(i \gg 1) + ((i \gg 3) \ll 2)$ (as is done using bit-shift calculation). If i is odd, then this calculation is just offset

by 4. This works, because the shifting makes such that S' contains series of the first 4 even indices of the original set, then the first 4 odd indices, then the next 4 even indices and so on. The $(i \gg 3)$ calculates the index of which of these series that i is found in, and by shifting it 2 bits to the left, we effectively multiply this index by 4 to find which series that our number is in. Then we calculate which of these 4 indices in the series that correspond to our number by calculating $(i \gg 1)$. If i is odd, we will have to add 4, since we are interested in the odd indices. □

If we were to use a word size that is large enough such that lines 17-19 get run again, then we would have to find a more complex computation to find the correct index. This will only happen when $d \geq 16384$, so we can safely assume that this will not be relevant in the near future unless new computers get invented with such large word sizes.

3.4 Time Complexity Analysis

We can now move on to showing the time complexity of the algorithm. First, we can try to find the running time of the first for-loop. At each iteration i, we will describe the amount of elements in $S^{(i)}$ as $n^{(i)}$ with $n = n^{(0)}$. We can then describe $n^{(i)}$ like so:

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

which means that the first for-loop in total takes

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

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

$$\begin{aligned} & O(n) + \sum_{i=2}^{\log_2(d)} \lceil \frac{n}{2^{i-l(i)}} \rceil \\ &= O(n) + \sum_{i=2}^{\log_2(d)} \lceil n 2^{l(i)-i} \rceil \\ &\leq O(n) + \sum_{i=2}^{\log_2(d)} (n 2^{l(i)-i} + 1) \\ &\leq O(n + \log(d)) + n \sum_{i=2}^{\log_2(d)} 2^{l(i)-i} \end{aligned}$$

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:

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

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

4 Empirical Evaluation

4.1 Methods

For the rest of the project, the main focus will be on showing the practical feasability of the parallel bit-counting algorithm compared to simpler algorithms. Furthermore, a comparison will be made to the `popcnt` CPU instruction that is implemented on most x64-86 CPUs and has the same purpose [16].

4.1.1 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 execution time 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[2] in mind. This has not been considered within the scope of this project however, but instead lends itself as an open research problem.

4.2 Implementation

4.2.1 Technology

The implementation to be measured upon has been written in Rust, a memory safe, fast, compiled programming language[14] 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[10].
The binary to be benchmarked is compiled and run on a laptop workstation as described in appendix A. The code for the implementation is available in appendix B.

4.2.2 Design

One of the main design challenges of the implementation is to support multiple word sizes simultaneously to give an accurate estimate on how the execution time scales with the word size. This has the consequence of making it harder to hard-code values like $m_{i,j}$ and $l(i)$, since a theoretical implementation with a very large word size would have to compute these values at runtime.
This implementation uses hard-coded values however, since any real-life implementation with a reasonable word-size would do so as well.
Another challenge is code optimization. Rust has an `#[inline(always)]` macro that forces function inlining, which reduces the time needed to handle the stack. A nice feature about the algorithm is that it does not use multiplication, which usually is a very slow operation. It does however operate on very large lists of numbers, which need to be handled efficiently. For this, the heap-allocated `Vec` data structure was used, as it functions as a dynamic array and contains many features to help with optimizations. For instance, the `Vec::with_capacity(n)` constructor specifies an initial capacity of the dynamic array such that unnecessary allocations are avoided, and the `Vec::truncate()` allows constant time shortening of arrays[14].

4.2.3 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.
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.

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

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.

4.2.4 Benchmarking

To benchmark the algorithm, two reference implementations were created to be used as a baseline models alongside the CPU instruction. 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 of the algorithms.
The Rust library `Criterion.rs`[12] 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 execution time of a single operation - instead it runs multiple samples each consisting of a number of iterations. The running time is then calculated as the execution time per sample divided by the amount of iterations.

4.3 Results

The benchmarks were run on different input sizes that were all powers of two - in this case $\{2^4, 2^5, \ldots 2^{19}\}$ and with word sizes $\{16, 32, 64, 128\}$. The results can be seen in figure 1. It can be seen that the parallel algorithm is significantly slower for small word sizes, and only becomes more efficient than the naïve sequential algorithm when $d = 128$. This must mean that the hidden factors hidden in the big-O notation of the parallel algorithm's time complexity outweigh the d factor of the naïve algorithm. This makes sense, since the parallel algorithm is much more complex and only starts benefitting from parallelism after the first three iterations. Interestingly enough, when $d = 64$, the parallel algorithm only seems to be slower when the input size is larger than 2^{16}. This can be due to the input data being able to fit inside 128 KiB Li1 data cache on my machine (see appendix A for machine specifications).
When run on 128-bit size words, the naïve sequential algorithm performs much worse than than both the parallel and the divide-and-conquer algorithm. This shows that when we increase the word size, algorithms that have a linear dependency on d suffer much more than algorithms that have a logarithmic dependency, which is as expected. It is also seen that the divide and conquer algorithm significantly outperforms both the parallel and the naïve algorithm at all word sizes. This is might be due to the fact that it is much simpler than the parallel algorithm and is asymptotically more optimal than the naïve sequential algorithm. The final thing to note is that the `popcnt` CPU

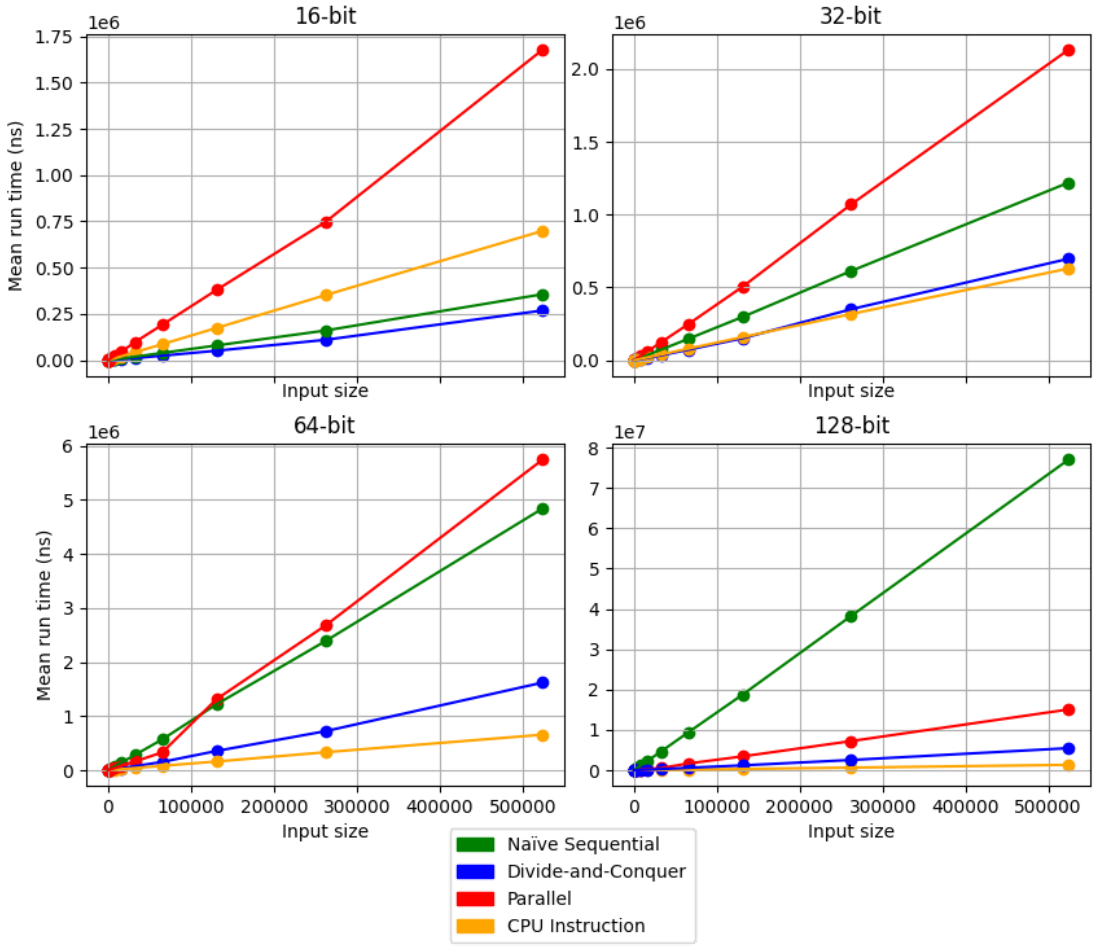

Figure 1: Execution time of each algorithm on different word-lengths

instruction is the ultimately fastest option out of the algorithms. This shows that these algorithms might only be useful on hardware that does not support this instruction or on very large wordsizes. Even then, the divide-and-conquer algorithm seems to be sufficiently efficient for many uses or even superior to the parallel algorithm despite its suboptimal theoretical running time.

4.4 Discussion

Even though the results indicate a clear advantage to simpler algorithms to perform bit counting, the parallel bit counting algorithm should not be completely dismissed. The experiment does not tell anything about how the algorithm performs on very large word size (e.g. 256 to 384 bit words) as seen on some GPU units[13]. Since the applications of the similarity search problem usually apply to very large data sets, it is often appropriate to perform highly efficient parallel calculations when possible. Parallel programming comes with many new challenges that were not considered in this project such as minimizing read/write operations to the main memory of the computer, the memory capacity of the graphics card and the size of the register files. This makes an actual implementation of the algorithm 3 non-trivial, especially since most programming interfaces for interacting with a GPU relies on intricate low-level systems programming. A language like Futhark might ease this however[15].
Another thing that this experiment does not take into account is the index calculation time required to actually look up into the list of cardinalities. This was not included, since the correctness criteria of the algorithm may or may not take into account whether or not the cardinalities should have the same index in the final list as the input data. A general formula for this was not found, so the calculation time cannot be shown to be asymptotically positive in regards to the word size (although one might expect it to be), which makes any results unconclusive to real life applications anyways.
The implementation that this experiment relied upon was optimized as per the best of my abilities, but futher optimizations might be possible. A nice feature of the Rust programming language is that the time complexity is very often documented in the documentation, which was very useful when working with heap-allocated vectors of numbers. The implementation was also written to be as generic as possible in regards to word size such that the experiment would be performed as unbiased as possible. This was possible due to Rust's generics, and while Rust design philosphy relies upon zero-cost abstractions[14] (like C++), this might cause some optimization issues. Further optimizations might also be possible by manually analyzing the produced assembly code, which was regarded as out-of-scope for this project.
In general, while parallel bit counting theoretically is more efficient than its sequential and instruction-level counterparts, it has yet to be shown to be the case in practice. While further research might research how it performs on specialized hardware, the algorithm behind this will probably first be useful when larger word-sizes become more easily available and computable, if ever.

5 Conclusion

An in-depth analysis of existing solutions to the Approximate Similarity Search Problem described how many existing solutions utilize sketching to achieve a low query time with a constant error probability. In the case that one wants a subconstant error probability, a possible solution by Knudsen [11] was presented that can reduce the query time even further.

An important part of this optimization relies in being able to quantify the accuracy of a collection of datastructures very efficiently, which is done in part by calculating the cardinality of a list of bit-strings. Knudsen [11] presents an algorithm to do this in parallel, which reduces the total query time to be sub-linear to the amount of sets in the corpus.

An analysis of the bit-counting algorithm shows its correctness with a few modifications alongside a theoretical running time which matches [11]. Benchmarks performed on an actual implementation tells otherwise however, showing that the word parallelism used to achieve a low amortized cost actually is hard to match simpler algorithms in practice when used on standard word-sizes. Further research might focus on benchmarking the algorithm on more specialized hardware, although rewriting the implementation seems to be non-trivial.

Mathematical Notation

Although sets by definition are not ordered, it is commonly seen in the litterature to index into a set (see [11] for an example). Therefore, sets are regarded as ordered for this project.
Expressions within square brackets are indicator variables denoted using Iverson notation:

$$[X = Y] \in \{0, 1\}$$

$$0 \leq Pr[[X = Y] = 1] \leq 1$$

Integers within square brackets denotes the integer interval between 0 and the number.

$$[n] = [0 \ldots (n-1)] = \{0, 1, \ldots, n-1\}$$

Numbers in a different base than 10 will be indicated with a subscript.

$$11010110_2 = 214$$

Bit-shifting is done using C-style notation. Therefore

$$0101_2 \ll 1 = 1010_2$$

$$1010_2 \gg 1 = 0101_2$$

Logical operations between integers are meant as bit-wise.

$$\text{AND:} \quad 0101_2 \wedge 0011_2 = 0001_2$$

$$\text{XNOR:} \quad 0101_2 \odot 0011_2 = 1001_2$$

For this project, it is often useful to reference the same element after different iterations of an algorithm. To do this, the notation $x^{(i)}$ means an element after i iterations of an algorithm.

Appendices

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

B Code

The code for the implementation part can be found on in this Github repository: `https://github.com/Snailed/parallel-bit-counting`.

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] A. Andoni and P. Indyk, "Efficient algorithms for substring near neighbor problem," Jan. 2006, pp. 1203–1212. DOI: 10.1145/1109557.1109690.

[5] 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.

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

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

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

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

[10] P. Emmerich, S. Ellmann, F. Bonk, *et al.*, "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.

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

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

[13] *Nvidia geforce rtx 3090 specs*, https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622, Accessed: 2021-12-22.

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

[15] *The furthark programming language*, https://futhark-lang.org/, Accessed: 2021-12-22.

[16] *The popcnt instruction*, https://www.felixcloutier.com/x86/popcnt, Accessed: 2021-01-05.