Parallel Bit-Counting for Approximate Similarity Searching

Rasmus 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[7]. 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

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 $J(A, Q) = \frac{|A \cap Q|}{|A \cup Q|}$[5][7]. Practical applications includes searching through large corpi of high-dimensional text documents like plagiarism-detection or website duplication checking among others[6]. 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.* [4] 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*[3], *fast similarity sketching*[5] and *parallel bit-counting*[7] (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[2]. 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 [7]): 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.* [4] 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 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 $\frac{|V \cap W|}{|V \cup W|} = J(V, W)$ □

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 $S(A) = \langle H_0(A), \dots, H_{k-1}(A)\rangle$. 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 Fast Similarity Sketching

2.5 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 [7] 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

```
function CARDINALITY($w$, $d$)    ▷ $w$ is the input word, $d$ is the word-size
    $x \leftarrow 0$
    $i \leftarrow 0$
    while $i \leq d$ do
        $x \leftarrow x + (w \gg i) \wedge 1$
        $i \leftarrow i + 1$
    end while
    return $x$
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 [7] 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.

2.5.1 Divide-and-Conquer

To explain how divide-and-conquer methods can be used to improve run-time, we must first introduce a bit mask from [7] defined like so:

$$m_{i,j} = \underbrace{0\ldots0}_{2^j-2^i}\underbrace{1\ldots1}_{2^i}\ldots\underbrace{0\ldots0}_{2^j-2^i}\underbrace{1\ldots1}_{2^i}$$

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

```
function CARDINALITY($w$, $d$)    ▷ $w$ is the input word, $d$ is the word-size
    $i \leftarrow 0$
    while $i \leq \log_2(d)$ do
        $w \leftarrow T(w, m_i, 0) + T(w, m_i, 2^i)$
        $i \leftarrow i + 1$
    end while
    return $w$
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 pre-computed. 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.

2.5.2 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 $t_0, \dots, t_{2^{i-l(i)}-1}$ 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. □

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 $O(m + \log(d))$.

2.6 Comparison

3 Methods

3.1 Hypothesis

3.2 Benchmarking

3.3 Expected Results

4 Implementation

4.1 Technology

4.2 Design

4.3 Assumptions

4.4 Challenges

4.5 Correctness

4.6 Benchmarking

5 Results

6 Discussion

7 Conclusion

References

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

[2] 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.

[3] 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.

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

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

[6] S. Vassilvitskii, *Coms 6998-12: Dealing with massive data (lecture notes, columbia university)*, 2018.

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