A Fast Sketch-based Assembler for Genomes

Priyanka Ghosh
School of EECS
Washington State University
Pullman, WA 99164
pghosh@eecs.wsu.edu

Ananth Kalyanaraman
School of EECS
Washington State University
Pullman, WA 99164
ananth@eecs.wsu.edu

ABSTRACT

De novo genome assembly describes the process of reconstructing an unknown genome from a large collection of short (or long) reads sequenced from the genome. A single run of Next-Generation Sequencing (NGS) technologies can produce billions of reads, making genome assembly computationally demanding. One of the major computational steps in modern day short read assemblers involves the construction and use of a string data structure called the *de Bruijn graph*. In fact, a majority of short read assemblers build the complete de Bruijn graph for the set of input reads, and subsequently traverse and prune low-quality edges, in order to generate genomic "contigs" — the output of assembly. These steps of graph construction and traversal, contribute to well over 90% of the runtime and memory. In this paper, we present a fast algorithm, *FastEtch*, that uses sketching to build an approximate version of the de Bruijn graph for the purpose of generating an assembly. The algorithm uses Count-Min sketch, which is a probabilistic data structure for streaming data sets. The result is an approximate de Bruijn graph that stores information pertaining only to a selected subset of nodes that are most likely to contribute to the contig generation step. In addition, edges are *not* stored; instead that fraction which contribute to our contig generation are detected on-the-fly. This approximate approach is intended to significantly improve performance (both execution time and memory footprint) whilst possibly compromising on the output assembly quality. For further scalability, we have implemented a multi-threaded parallel code. Experimental results using our algorithm conducted on *E. coli*, *Yeast*, and *C. elegans* genomes show that our method is able to produce assemblies with quality comparable or better than most other state-of-the-art assemblers, while running in significantly reduced memory and time footprint.

Categories and Subject Descriptors

J.3 [**LIFE AND MEDICAL SCIENCES**]: Biology and genetics; E.1 [**DATA STRUCTURES**]: Graphs/networks

BCB'16, October 02-05, 2016, Seattle, WA, USA

DOI: http://dx.doi.org/10.1145/2975167.2975192.

General Terms

Algorithms, Bioinformatics

Keywords

Genome assembly, de Bruijn Graph, Count-Min sketch, Approximation methods

1. INTRODUCTION

De novo genome assembly is the problem of assembling an unknown genome from a set of DNA "reads" sequenced from it. Despite advances in assembly algorithms over the last two decades, development of *de novo* genome assembers continues to be an active research topic. Over the years, multiple Next Generation Sequencing (NGS) technologies have emerged (e.g. Illumina, 454), which have increased in throughput, improved in accuracy and decreased in cost, leading to a massive scale adoption of these technologies by practitioners. Most of these technologies generate "short reads" that are of length in the hundreds of bases, with errors less than 1%, but at rates that can quickly overwhelm the computational capacity to assemble them.

There are numerous short read assemblers that have been developed over the last decade (e.g., [1, 2, 11, 16, 17]). Nearly all of them use a string data structure called the *de Bruijn graph* to compute their assembly, and follow a 3-step approach: First, the de Bruijn graph is constructed using all the substrings of length k (called k-mers) in the input reads as "vertices", and all the $k+1$-mers that exist in the input reads as "edges" (see Fig. 1 for an example). The next step involves error correction, which locates and prunes paths induced by erroneous k-mers that are likely to be manifested by sequencing errors. The final step is to generate a set of assembled *contigs* by traversing paths in the residual graph. Of the three steps, the first step of de Bruijn graph construction is typically the most memory- and time-intensive, potentially taking hours to even days for assembling moderate sized eukaryotic genomes, and requiring tens to hundreds of gigabytes of memory.

To mitigate the memory requirements, a handful of recently developed assemblers [3, 14] use a probabilistic data structure (Bloom filter [10]) during their de Bruijn graph construction. The filter enables a succinct representation for finding and locating all k-mers that exist at the price of reporting false positives with a low probability. However, the Bloom filter is suited for membership queries. In the case of genome assembly, in addition to membership, the frequency of k-mers is also important. Therefore, the Bloom filter-based methods follow an indirect approach of first in-

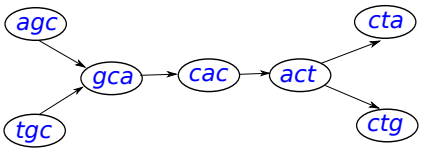


Figure 1: Example of a de Bruijn graph for an input containing the reads $\{agcactg, tgcacta\}$, and for $k = 3$.

dexing the k-mers using the filter and later pruning those k-mers (vertices) from the graph that have low frequency. Furthermore, it is essential to estimate the size of the filter *a priori* in order for it to perform optimally [7].

Contributions: In this paper, we present a new approach to compute genome assembly using a probabilistic data structure specifically designed for counting — viz. the *Count-Min sketch* (CM sketch) [6]. The data structure was originally designed for streaming applications and operates in sub-linear space. We propose a new genome assembly algorithm that uses the CM sketch to build a low memory, *approximate* version of the de Bruijn graph, and uses that graph to generate an assembly. More specifically, our approximate de Bruijn graph stores, with high probability, only those vertices corresponding to highly frequent k-mers. Edges are *not* stored — instead only a subset is detected on-the-fly, as contigs are generated. By avoiding the construction of a full blown de Bruijn graph and by selectively enumerating edges (and hence, paths) on-the-fly, our approach trades off assembly accuracy in favor of performance (both memory and run-time). Toward further scalability, our approach also parallelizes efficiently on multi-core computers. Experimental results using our algorithm conducted on *E. coli*, yeast, and *C. elegans* genomes show that our method is able to produce an assembly with quality comparable or better than most other state-of-the-art assemblers, while running in significantly reduced memory and time footprint. Henceforth, we call our algorithm *FastEtch*, referring to its fast ability to "etch" out a genome using sketching.

2. RELATED WORK

De novo genome assembly is a well researched topic with a number of assembly tools and strategies developed over the last two decades. Short read assemblers correspond to that subset which target reads generated from NGS technologies (e.g. Illumina, 454 pyrosequencing, SOLiD). Broadly speaking, modern day short read assemblers can be grouped under three categories [12]: i) *Overlap-Layout-Consensus (OLC)* methods that rely on constructing a pairwise alignment based overlap graph; ii) *de Bruijn Graph (DBG)* methods that build a de Bruijn graph out of the k-mers occurring in reads, and then perform error correction and Euler path traversals to generate an assembly; and iii) *String Graph* approaches that represent a variation of the OLC approach, by focusing on suffix-prefix overlaps between reads.

Of these approaches, the DBG approach has emerged to be one of the most widely used paradigms. Originally introduced by Pevzner *et al.* [13], it has been widely adopted in a number of short read assemblers including (but not limited to): Velvet [17], ALLPATHS [1], SOAPdenovo [11], and ABySS [16]. These methods vary in the manner in which they process the de Bruijn graph, and identify and prune potentially erroneous paths during contig assembly. However all of these assemblers follow the same algorithmic template:

1. (*Graph Construction*) Given a set of input reads and a constant $k > 0$, construct the de Bruijn graph;
2. (*Error Correction*) Identify and prune off paths in the de Bruijn graph that are likely to be an artifact of sequencing errors and inconsistencies; and
3. (*Contig Generation*) Compute path traversals of the residual graph to generate contig assembly.

The DBG approach has its performance advantages — constructing a de Bruijn graph is linear in time complexity (in comparison, building an overlap based graph is quadratic). In addition, the traversal step treats the problem of contig generation as an Euler tour (which is polynomial solvable) as opposed to the intractable Hamiltonian path problem as OLC approaches do. Despite this performance advantage, assembling large, complex genomes still remains a compute- and memory-intensive task, requiring hours to days of computation, and tens to hundreds of gigabytes of memory.

In an attempt to reduce the memory footprint of assembly, Chapman *et al.* proposed the Meraculous algorithm [3]. The algorithm uses the Bloom filter, a probabilistic data structure for membership queries, to generate the de Bruijn graph. Once the graph is generated, simple chain paths (between branching nodes) are enumerated and the first batch of unique contigs are generated from these paths. Subsequently, the read set is loaded multiple times (iteratively) to incrementally map the reads against the contigs and in the process elongate the unique contigs on either end.

Minia is another assembler that uses the Bloom filter [4] to construct the de Bruijn graph, by storing all the distinct k-mers (vertices) and subsequently discarding the ones considered potentially erroneous (appearing fewer times than a threshold). The tool also stores an additional structure to remove what the authors call "critical false positives". Moreover, in order to further reduce memory footprint, the Minia algorithm uses secondary storage. All k-mers generated from the read set are partitioned and stored on the disk. Low-abundance k-mers are filtered, by separately loading each partition (one at a time) into memory and into a temporary hash table. In the absence of adequate disk space, we observed Minia to encounter segmentation faults.

There have been other efforts to generate memory-efficient, compressed representation of the string structure used for assembly, viz. bitmaps [5], and FM-index [15].

The algorithm presented in this paper, *FastEtch*, differs from all the above previous efforts in the following ways: i) It is the first approach, to the best of our knowledge, to use the Count-Min Sketch (a *sublinear space* data structure) in the graph construction step; ii) It constructs an *approximate de Bruijn graph*, storing only a *subset* of vertices (k-mers), and detecting edges on-the-fly as contigs are generated — thereby saving on *both* vertex and edge space requirements, and consequently, the time to solution; and iii) Through a choice of parameters, it offers a way to control the quality-performance (time, memory) trade-off in the output.

3. METHODS

3.1 Notation and Terminology

Let r denote a read (string) of length ℓ, over the DNA alphabet $\{a, c, g, t\}$. We index the characters in each read from 1. Let $r(i, l)$ denote the substring of length l starting at index i in r, such that $i + l - 1 \leq \ell$. Then, every $r(i, k)$ is a *k-mer* in r, for a fixed $k > 0$. We denote the input set

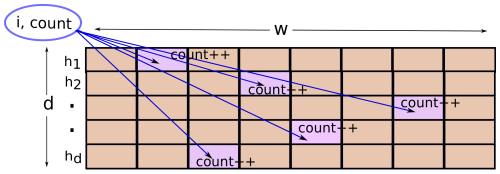


Figure 2: CM sketch depicting the *Update* function, which updates the count for every k-mer to one cell in each row.

of n reads as $\mathcal{R} = \{r_1, r_2, \ldots r_n\}$, and their total length as N $(= \sum_i |r_i|)$. Let $\mathcal{K}$ denote the set of all k-mers in $\mathcal{R}$.

Given $\mathcal{R}$ and k, the corresponding *de Bruijn* graph is a directed graph $G(V, E)$, such that the vertex set $V = \mathcal{K}$, and there exists a directed edge $(u, v) \in E$ for every pair of k-mers u and v that are consecutive in any read $r \in \mathcal{R}$ (as shown in Fig. 1). We note here that the above represents a minimalist definition of a de Bruijn graph. In practice, typically a number of other attributes are stored along each edge — e.g., the set of reads that cover each edge (or vertex), or the number of occurrences of every k-mer.

3.2 Sketching for Genome Assembly

Introduced in 2003, the Count-Min sketch (CM sketch) [6] is a sublinear space data structure used for summarizing massive data streams in applications. It finds use in implementing point, range and inner product queries, quantile computations, and identification of heavy hitters [6]. The *CM sketch* data-structure is a 2-D matrix of depth d and width w, that stores $d \times w$ counts (Fig. 2). The sketch uses d hash functions:

$$h_1 \ldots h_d : \{1 \ldots n\} \rightarrow \{1 \ldots w\}$$

that are from a pairwise independent family. Two parameters, ε and δ, are used to determine the sizes of w and d, where ε denotes the error in answering a query within the probability of 1 - δ: $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$.

CM sketch supports basic functions: *Update count* and *Estimate count*. In the streaming model, every incoming update is of the form $< i, c >$, where i represents an item and c represents the count of i for that update. Given an update $< i, c >$, the *Update count* function is given by:

$$count[j, h_j(i)] \leftarrow count[j, h_j(i)] + c, \ \forall j \in [1, d] \qquad (1)$$

The *Estimate count* function retrieves an estimated count for a particular item i from the CM sketch, and is given by:

$$CM(i) \leftarrow \min_j \quad count[j, h_j(i)] \qquad (2)$$

Intuitively, if a_i denotes the actual count for item i, then each entry $count[j, h_j(i)]$ represents an overestimate for a_i (due to potential collisions introduced by h_j). Therefore, returning the minimum over all the d counts represents the lowest overestimate one can derive from the CM sketch.

Given that $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the CM sketch provides the following approximation guarantee [6]:

$$CM(i) \leq a_i + \varepsilon \sum_{i'} a_{i'}, \text{ with probability at least } 1 - \delta \quad (3)$$

As can be observed from this approximation bound, the quality of the estimate is likely to be better for the more frequent items in the set — i.e. those items whose a_i is a relatively large fraction of $\sum_{i'} a_{i'}$. This makes the CM sketch naturally suited for keeping track of frequent items in data streams.

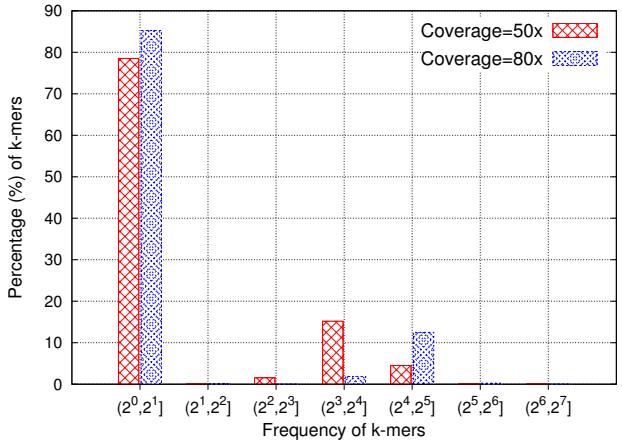


Figure 3: Histogram showing k-mer frequency distribution (expressed as a % of the total distinct k-mers), for two coverage experiments of the *E_coli* genome, with k=32.

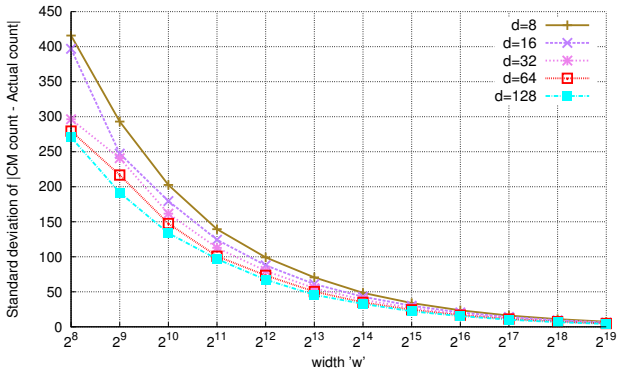
3.2.1 *Frequency of k-mers*

In this paper, we describe a way to make use of the CM sketch for detecting frequent k-mers in reads and using them for constructing an approximate version of the de Bruijn graph, and subsequently, for generating the assembly. Our idea to detect and keep track of only frequently occurring k-mers is directly motivated by *sequencing coverage* C — which corresponds to the number of clonal copies of the target genome used in sequencing. Typically, *de novo* sequencing experiments use a high coverage to sequence the underlying genome (between $> 50\times$ and $100\times$). Given the stochasticity of the random shotgun process, this implies that on an average, each base (and hence, each k-mer originating at that base) in the genome is covered by roughly C reads. However, errors during sequencing, even if as low as 1%, alter this expectation, as erroneous positions in reads generate low frequency ("*poor quality*") k-mers. In fact, the fraction of such low quality k-mers is only expected to grow as k is increased. The non-erroneous fraction ("*high quality*" k-mers) on the other hand tend to have a frequency proportional to the coverage.

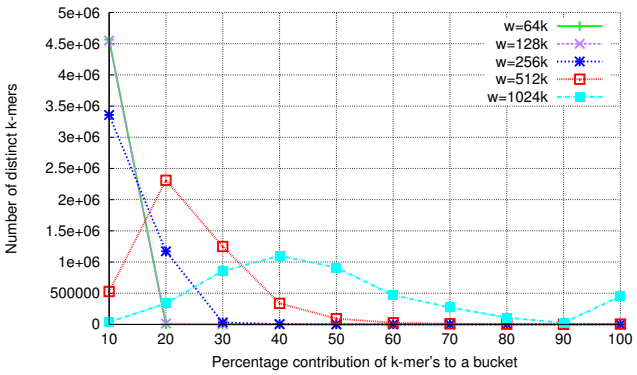
Consequently, if we are to plot the frequency distribution of k-mers, we expect to see a bimodal distribution, with roughly two peaks, one corresponding to the low quality k-mers and the other high quality k-mers. We conducted a preliminary study on the frequency distribution of k-mers generated from two different coverage experiments on the *E. coli* genome. Fig. 3 confirms the expectation.

In fact, this chart also provides an interesting insight into the relative concentrations of the low vs. high quality k-mers. Notably, well over 75% of all the distinct k-mers are of low quality, while only up to 25% are high quality k-mers. This simple observation lays the foundation for our use of CM sketch for genome assembly — if CM sketch can be used to effectively *identify only those high quality k-mers*, then significant savings in space and time can be achieved.

Fig. 3 also suggests that there exists a clear separation between low and high qualilty k-mers in their frequency ranges



(a) CM sketch: precision



(b) CM sketch: bucket contribution

Figure 4: The effect of w and d on: a) the standard deviation of the difference between CM count and actual count for all k-mers. b) the percentage contribution of each k-mer to their respective hash bucket in the CM sketch.

— i.e. as per the bimodal expectation. We exploit these observations in the design of our algorithm (Section 3.3).

3.2.2 *Precision of CM Sketch Counts*

In this section, we evaluate the precision of CM sketch counts when applied to k-mers from sequenced reads. As described earlier, the size parameters w and d of the CM sketch play a significant role in controlling the quality of the count overestimate provided by the sketch. Also note that the summation $(\sum_{i'} a_{i'})$ in the approximate bound (3) will be $\Theta(N)$, if CM sketch is used for k-mer counting.

Given the above, we first note that even for small values of d it is possible to achieve a very high probability $1-\delta$ in bound (3). For example, $d = 5$ is sufficient to guarantee 0.99 probability (for $1-\delta$).

As for w, note that it controls the contributing factor ε of the summation $(\sum_{i'} a_{i'})$ into the estimate. This implies that high precision can be achieved by targeting a very low value of ε. But how small should ε be so as to make the contribution from the summation practically negligible, when applied to k-mer counting? For instance, a value of w as high as 10^6 will bring ε down to 10^{-6}. Assuming N is 10^9 (1 Gbp input), this still implies a non-negligible contribution from 10^3 for each estimate. Given that the sequencing coverage is typically only under 100×, this suggests that the value of w has to be increased proportional to N if one were to accurately recover the counts for *all* k-mers. However, such a high value of w is clearly undesirable from a memory consumption perspective, because as w tends to N, the sublinear space advantage of the CM sketch diminishes.

This is where our strategy of targeting only frequent k-mers (high quality) has a direct impact on the practical value of the CM sketch for k-mer counting. More specifically, since our goal is *not* to recover all k-mers but only those that are frequent, a smaller value of w to justify the sublinear argument is sufficient in practice.

To illustrate this, we performed an experiment with a 10× read set of the *E. coli* genome. For each distinct k-mer, we computed the difference between the estimated count (CM count) generated by the CM sketch and the k-mer's actual count. Fig. 4a shows the results of this comparison for varying values of w, and also for a smaller range of d. As shown, w has a more pronounced effect on the precision of the CM sketch compared to d. More importantly, the observations confirm that a small value of w (e.g. $128K$) is sufficient to diminish the standard deviation in the overall difference between actual and estimated (for all k-mers). Note that by contrast, the value of N is at least two orders of magnitude larger ($\approx 50M$ for this set). We also noticed that an increase in the value of d, does not significantly impact the output and therefore we have restricted the value of d to 8 in all our experiments.

We also performed an experiment to study the effect of varying w on the nature of collision within each hashed entry ("bucket") of the CM sketch. Intuitively, the more skewed a bucket is in its composition, the easier it is to separate the high quality k-mers from the rest. To study this effect, we conducted experiments for varying values of w, and calculating the percentage contribution of all distinct k-mers within their corresponding bucket[1]. This is obtained by dividing the actual count of each k-mer by its corresponding CM count. Fig. 4b shows the results of this analysis, as a histogram of k-mers distributed by their contributions to their respective buckets. As can be observed, as the value of w is increased, a larger fraction of k-mers contribute dominantly to their respective buckets — as indicated by the larger area under the curve for >60% contribution. This is because of lower collision rates. However, too large values of w implies more space cost for the CM sketch — a tradeoff between space and filtering efficacy.

3.3 *FastEtch:* The Assembly Algorithm

Building on the foundations laid out in Section 3.2, we present our *FastEtch* algorithm for genome assembly (see Fig. 5). Our algorithm has two steps:

i) (*Graph Construction*) Construct an "approximate de Bruijn graph" using the input reads and CM sketch;

ii) (*Contig Generation*) Generate contigs by performing edge detection and path enumeration of the approximate de Bruijn graph.

3.3.1 *Approximate de Bruijn Graph Construction*

Our graph construction algorithm uses two data structures: a $d \times w$ CM sketch, and an auxiliary data container to hold the output "approximate de Bruin graph" (denoted by $\hat{G}$), which only contains a selected subset of k-mers that have been identified as "high quality" by our method. Ide-

[1]Of the d buckets that a k-mer is present, the bucket with the minimum count for that k-mer is selected.

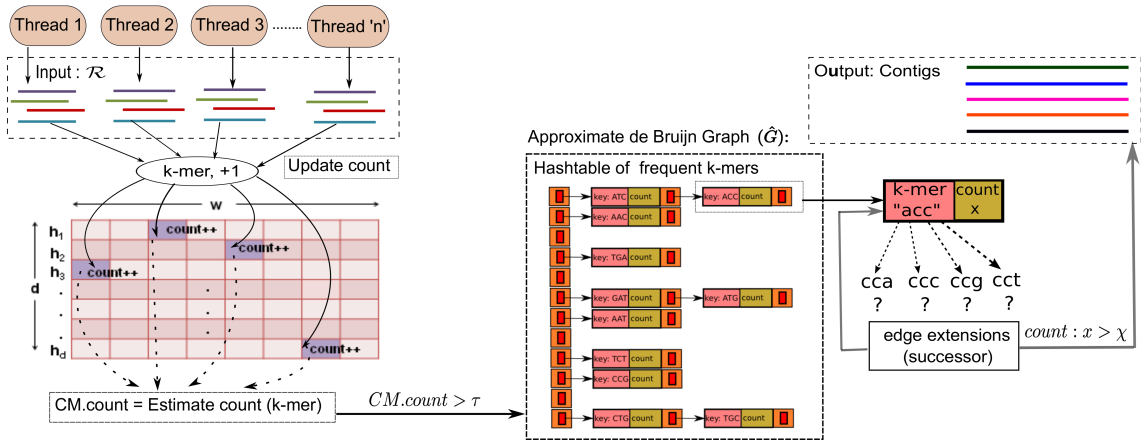


Figure 5: *FastEtch:* A schematic illustration of our sketch-based algorithm for *de novo* genome assembly.

Algorithm 1: Approximate de Bruijn Graph Construction Algorithm — *Baseline*

Input: Input set of reads: R, Width: w, Depth: d, Threshold: τ
Output: Approximate de Bruijn graph ($\hat{G}$)
for *each* $r \in R$ *in parallel* **do**
  for *each k-mer* $i \in r$ **do**
    for *each* $j \leftarrow 1$ *to* d **do**
      Update $count(j, h_j(i)) \leftarrow count(j, h_j(i)) + 1$
    Estimate $CM(i) \leftarrow \min_j \quad count[j, h_j(i)]$
    if $CM(i) > \tau$ **then**
      if $i \notin \hat{G}$ **then**
        Insert i in $\hat{G}[h(i)].list$
        Initialize $\hat{G}[h(i)].partialCount(i) \leftarrow 0$
      $\hat{G}[h(i)].partialCount(i)$++
  return $\hat{G}$

ally, these identified k-mers should correspond to all and only those frequent k-mers. For the purpose of the auxiliary k-mer data container, we use a simple hash table in our current implementation[2]. We use multi-threading for parallelism. Let p denote the number of threads.

Algorithm 1 presents our *baseline algorithm* to construct the approximate de Bruijn graph. Given the input set of n reads $\mathcal{R}$, the reads are partitioned in a load balanced manner (using dynamic scheduling) among the p threads. Each thread then enumerates all k-mers from its batch of reads, by simply sliding a window of length k within each read. For each k-mer i, the following operations are performed:

1) First, the k-mer's count is incremented in all the corresponding d buckets of the CM sketch, using the *Update count* function.
2) Next, we check to see if the k-mer's estimated count has exceeded a certain threshold τ. If it has, then that k-mer is deemed *tentatively* as "high quality" and is inserted into $\hat{G}$ (if it already does not exist). The first time a k-mer is inserted into $\hat{G}$, we initialize a *partial count* for this k-mer and set it to **0**. If the k-mer already exists in $\hat{G}$, then we simply increment the partial count.

[2]Note that this can be later extended to more compact representations such as the Bloom filter.

The rationale for maintaining a partial count for every distinct k-mer pushed into $\hat{G}$ is as follows: Ideally, the set of k-mers inserted into $\hat{G}$ should correspond to all and only those k-mers occurring above a certain frequency (set proportional to sequencing coverage C). However, due to collision of k-mers within each bucket of the CM sketch, it is possible that the estimated CM count for a k-mer, which is an overestimate, is significantly larger than its actual count. This could happen in particular within the CM sketch buckets where there is a skewed mixture of a handful of high frequent k-mers alongside numerous low frequency k-mers. If it so happens that multiple instances of even one of those high frequency k-mers are inserted early on, then by virtue of collision, any low frequency k-mer that follows into the bucket is also likely to get selected for insertion into $\hat{G}$. It is precisely for this reason that we deem any k-mer inserted into $\hat{G}$ as being *tentatively high quality*, and maintain a *partial count* for such k-mers in $\hat{G}$. By initializing this partial count to **0** (and *not* τ or the CM count value), we are essentially making this partial count stored at $\hat{G}$ an **underestimate** of the actual count. By sheer probability of occurrence, the low frequency k-mers inserted into $\hat{G}$ are likely to remain close to this initialized value, whereas the high frequency k-mers (the real high quality ones) are likely to grow in their partial count values, as more instances of that k-mer appear. This strategy of *moving from an overestimate (CM count) to an underestimate (partial count)* is critical to ensure that the k-mers that eventually get selected for contig generation (elaborated in Section 3.3.2) are indeed of high quality.

Improving the Efficacy of Filtering: Note that in the above baseline approach, once a bucket's CM count exceeds τ, all subsequent k-mers for which that bucket is the minimum count bucket, will be inserted into the hash table $\hat{G}$. But there is no guarantee that such k-mers are all indeed of high quality. In fact, too large a τ can negatively impact sensitivity (i.e. potentially missing out on a good fraction of high quality k-mer insertions). On the other hand, a low value of τ affects precision by letting in too many low quality k-mers into $\hat{G}$. Therefore, instead of solely relying on τ, we devise a "utility"-based iterative scheme to insert k-mers into $\hat{G}$, with the aim of keeping the number of insertions into $\hat{G}$ as low as possible without losing sensitivity. We define a measure called the *utility* for each CM sketch bucket, which will be measured dynamically with every CM sketch update

Algorithm 2: A Utility-based Scheme to Insert k-mers into $\hat{G}$

```
Input: CM sketch bucket: b, k-mer: i
Let i ← k-mer being inserted into bucket b, such that b
is the count-min bucket for i
Increment b.CMcount
if b.CMcount > τ then
    if b is of "high utility" then
        if i ∉ Ĝ then
            Insert i into Ĝ and initialize partial count
            Increment b.InsertCount
        else
            Increment partial count for i in Ĝ
        if reset criterion is met then
            Reset b.CMcount = b.InsertCount = 0
```

of that bucket. Intuitively, a CM sketch bucket b is said to be of *high utility* if it has a significant potential to contribute to a high quality k-mer into $\hat{G}$ at any stage.

To measure utility, within each CM sketch bucket, we keep track of an integer for the number of insertions into $\hat{G}$ that has resulted from this bucket so far. We refer to this count as the bucket's "*insert count*", and it is initialized to 0. This is kept in addition to the CM count of that bucket. Given the above, we provide two different ways to determine the utility of a bucket:

DEFINITION 1. *A bucket is said to be of* high utility *if its CM count has exceeded τ and its insert count is less than a certain threshold γ;*

DEFINITION 2. *A bucket is said to be of* high utility *if the ratio of its CM count to insert count is above a certain threshold β;*

The revised algorithm to insert a k-mer into $\hat{G}$ is given in Algorithm 2, using one of the two definitions of choice to compute a bucket's utility. Consequently, we generate two variants — *FastEtch-γ* and *FastEtch-β* respectively.

Intuitively, as per Definition 1, the idea is to prevent a bucket from performing too many inserts into $\hat{G}$ as that is indicative of low quality k-mers. This is achieved by placing a bound on the number of inserts (γ). However, if we permanently shut off a bucket's contribution after γ inserts are reached, then any new high quality k-mer that gets inserted after that point will also be inadvertently skipped. To reduce this loss of sensitivity, we *reset* the entire bucket by resetting both its CM count and insert count to 0, when a bucket is no longer of *high utility*. This process conducted iteratively for a bucket, lasts until all k-mers in the input are exhausted.

Definition 2 is similar in spirit except that the criterion for resetting a bucket's counts is based on the the ratio of its CM count to insert count. Intuitively, if a bucket holds too many low quality k-mers, this ratio will be close to 1. However, a higher value of the ratio presents a better evidence for high quality k-mers in that bucket.

3.3.2 Contig Generation

We use the approximate de Bruijn graph ($\hat{G}$) constructed as above to generate contigs. Algorithm 3 presents our algorithm to generate the assembled contigs. The main steps are as follows: Given a k-mer $i \in \hat{G}$, we define two functions:

Algorithm 3: Contig Generation Algorithm

```
Input: Approximate de Bruijn Graph: Ĝ
Output: Contigs
for each k-mer i ∈ Ĝ in parallel do
    if i is a begin k-mer then
        Initialize contig c ← first k − 1 characters of i
        repeat
            Concatenate the end character of k-mer i to
            contig c
            i ← succ(i)
        until i does not exist;
        Output contig c
```

$succ(i)$ and $pred(i)$. Note that i is a string of length k. A *valid successor* of i is a k-mer $i' \in \hat{G}$ (if exists) such that $i(2, k-1) = i'(1, k-1)$ *and* the partial count of i' is greater than a small constant χ (we used $\chi = 1$ in our experiments). Given that there are four such possible extensions of i, there could be at most four valid successors of i in $\hat{G}$. Of these, $succ(i)$ is set to that k-mer which has the largest partial count, as long as that partial count is greater than χ. The partial count check is essential to ensure that low quality k-mers which occur sparsely but which made their way into $\hat{G}$ get discarded during the contig generation step.

The notion of a *valid predecessor* and the $pred(i)$ function are defined similarly except that a predecessor needs to match its last $k-1$ characters with the first $k-1$ characters of i. These two functions provide the basis for contig generation. More specifically, we begin with any k-mer in $\hat{G}$ that does *not* have a predecessor. We refer to such k-mers as *begin* k-mers. From a begin k-mer we identify successors, one at a time, and elongate the contig until no further extension is possible. Consequently, this scheme constitutes our strategy to detect edges on-the-fly — not all but only those edges that contribute to contig assembly.

We keep a flag to mark k-mers that have already been used for generating a contig, so that it does not get selected as part of another contig. Furthermore, if during successor and predecessor operations, a given k-mer extension with the maximum partial count has already been claimed, then the next best choice is selected and returned as our heuristic. In our parallel implementation, we use *atomic updates* (as opposed to locks) to prevent multiple threads from updating the same k-mer entries concurrently.

3.3.3 Complexity Analysis

Graph construction: The time to generate k-mers from the reads and update the CM sketch is $\Theta(N)$. The insertion of the selected k-mers into the hash table for $\hat{G}$ depends on the collision rate at that hash table bucket. Our current implementation uses chaining, and if the k-mers are uniformly distributed across the hash table entries, then we can expect near constant time *per* hash table update as well. The graph construction step uses a combination of atomic updates and locks, to update the CM sketch and $\hat{G}$, and we expect the speedup of this step to be sublinear.

Contig generation: The time for this step is bounded by the total number of k-mers that were originally inserted into $\hat{G}$, which in turn can be no greater than the number of *distinct* k-mers in the input ($\mathcal{O}(N)$). In practice, though, we expect the size of $\hat{G}$ to be significantly smaller than N (as discussed in Section 3.2.1). Assuming perfect speedup from

Table 1: Input data sets used in our experiments

Organism	Genome Size (bp)	Coverage	# of Reads	Avg length	Data size (GB)
E_coli	4,703,541	80	3,713,280	100	0.4
Yeast	12,157,105	100	12,156,200	100	1.4
C.elegans	100,286,401	50	50,143,050	100	5.3

multi-threading using p threads, this implies an expected runtime complexity of $\Theta(\frac{N}{p})$. Since the contig generation step is lock-free we expect linear scaling in that step.

As for space complexity, the CM sketch is a sublinear data structure. The memory cost of our algorithm is dominated by the space to store $\hat{G}$. Note that $\hat{G}$ stores only those distinct k-mers identified by the sketch as tentatively high quality (as described in Section 3.3.1). Also, our algorithm does *not* need to store the input set of reads. Consequently, the expected space complexity is sublinear ($o(N)$).

4. RESULTS

4.1 Experimental Setup

We tested our assembler using read set inputs from three different organisms namely *E_coli* (K12_MG1655), *Yeast*, and *C.elegans* (shown in Table 1). All read data sets were generated using the ART Illumina read simulator [9] for different coverage settings. Experiments were conducted on a single 128GB DDR4 memory node of the NERSC Cori supercomputer (Cray XC40), with each node equipped with two sockets, each socket populated with a 16-core, 2.3 GHz Intel Haswell processor.

For comparative evaluation, we compared our assembler *FastEtch* with other state-of-the-art de Bruijn graph based short-read assemblers, viz. Velvet [17], SOAPdenovo [11], ABySS [16], and Minia [4]. We have presented results for *FastEtch-baseline* alongside two additional variants denoted by *FastEtch-*γ and *FastEtch-*β (described in Section 3.3.1). We used the QUAST [8] tool for quality evaluation, which reports the following metrics: N50 contig length (similar to a median contig length), % of genome covered, unaligned contig length (length of those contigs which have no alignment with the reference), and length of the largest region aligning with the genome.

4.2 *FastEtch*: Performance Evaluation

Table 2 presents the performance (runtime and memory) and quality statistics for *FastEtch-baseline*, for the three inputs. We also report on internal de Bruijn graph statistics, viz. the number of k-mers inserted into $\hat{G}$, and the number of k-mers that eventually contributed to contigs.

First we note that the number of k-mers that are inserted into $\hat{G}$ is one order of magnitude smaller than the input size (N). This shows the efficacy of CM sketch as a first level of filter. Subsequently, we note that there is a further one order of magnitude reduction in the number of k-mers that eventually get selected for contributing into the contigs. This shows the high selectiveness of our second filter ($\chi = 1$) in filtering out poor quality k-mers from $\hat{G}$. Such poor quality k-mers represent false positives as they were identified for insertion into $\hat{G}$ despite being infrequent. Section 4.3 presents results for further reducing these false positives. Put together, these statistics show that our *FastEtch-baseline* is able to lead to a 100-fold reduction in data complexity (from reads to contigs), for all three inputs.

Table 2: Performance evaluation of *FastEtch-baseline* across all three organisms, with k=32, using 32 cores

FastEtch-baseline	Ecoli cov=80x	Yeast cov=100x	C.Elegans cov=50x
Input size N (bp)	3.71×10^8	1.21×10^9	5.04×10^9
No. kmers in $\hat{G}$	5.6×10^7	1.8×10^8	8.7×10^8
No. k-mers in contigs	4.6×10^6	1.1×10^7	8.5×10^7
N50 (bp)	14,866	17,223	4,574
Coverage (%)	98.15	94.76	85.10
Largest alignment (bp)	83,580	98,269	94,576
Unaligned contig len. (bp)	3,494	11,290	640,307
Time (sec)	70.86	230.56	1,245.24
Memory (GB)	2.22	7.42	34.50

Table 3: Breakdown of execution time for different stages of assembly as seen for *FastEtch-baseline* for all the 3 inputs.

	Time in seconds			
Input	Reading	Graph Construction	Contig Generation	Total time
E_coli_80x	4.83	61.59	4.41	70.83
Yeast_100x	15.40	189.41	25.27	230.07
C.elegans_50x	41.56	795.27	407.90	1,244.73

As for quality, the results show that, despite the approximation of de Bruijn graph, the output quality of the contigs is maintained relatively high. The N50 contig lengths are in tens of thousands for both *E_coli* and *Yeast*, while for *C. elegans* the N50 length is in the thousands. Note that for *C. elegans* we used a lower sequencing coverage (50×). The above trend is also reflected in the overall genome covered by the contigs. The largest aligning regions in all three assemblies were substantially larger ($> 80K$) than the N50 contig length, indicating the ability of the assembler to capture long regions along the genome (via one or more contigs). The unaligned contig lengths are for those contigs which have no alignment against the reference, indicating false assemblies generated probably due to false extensions during contig generation. These unaligned fractions represent about 0.07%, 0.01% and 0.6% of the *E_coli*, *Yeast* and *C. elegans* genomes, respectively.

Table 2 also shows the raw performance (runtime and memory) of *FastEtch*. Table 3 shows the runtime breakdown by the major steps of the algorithm. As expected, the runtime is dominated by graph construction. In our experiments, we fixed the length of the hash table that stores $\hat{G}$ to roughly 14M buckets — significantly less than N in anticipation of a large fraction of k-mers getting filtered out. However, based on the actual number of k-mers inserted into $\hat{G}$, this hash table length implies an average collison rate of 3.61, 12.5 and 40 per bucket for the *E_coli*, *Yeast* and *C.elegans* datasets respectively. The length was fixed to reduce memory footprint but at the expense of runtime performance (since, due to multi-threading, a bucket needs to be locked prior to inserting a k-mer). Consequently, the length of the hash table represents a trade-off between runtime and memory usage. Furthermore, the choice of parameter τ has a significant impact on the number of k-mers that are inserted in $\hat{G}$, and thereby on the overall performance (both time and memory). Extended study of these parameters are presented in Section 4.3.

Fig. 6 shows the speedup for our parallel implementation for varying number of cores (threads). The contig generation step shows near linear scaling because it is lock-free. The

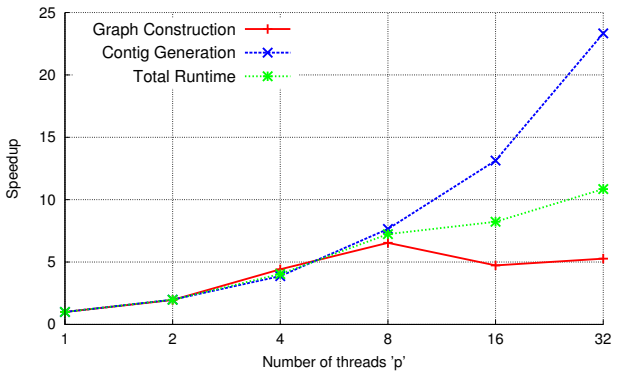


Figure 6: Speedup of *FastEtch* on *C.elegans* (50x, k=32).

γ	#k-mers in $\hat{G}$	N50 (bp)	% Genome Covered	Largest alignment (bp)
500	41,971,992	10,392	98.14	58,465
600	42,681,119	13,062	98.14	68,543
700	42,684,266	12,689	98.13	83,009
800	42,687,750	13,203	98.14	84,565
900	42,678,023	14,179	98.23	69,866
β	#k-mers in $\hat{G}$	N50 (bp)	% Genome Covered	Largest alignment (bp)
2	42,685,731	14,055	98.18	99,489
4	42,688,740	14,305	98.17	83,857
5	42,679,387	14,051	98.14	97,119
6	42,016,001	5,137	98.10	30,920

Table 4: Table comparing *FastEtch*-γ vs. β versions on *E_coli* dataset (Cov=80x) with k=32 and $\tau = 1,300$. Recall that N for this dataset $= 3.71 \times 10^8 bp$.

graph construction step scales only up to 8 threads. This step is affected by the collision rate — i.e. locks used to grant access to concurrently accessed buckets. The time (as well as memory) taken to update the CM sketch is negligible.

As for memory, the overall memory consumption is dominated by the space to store $\hat{G}$, which is implemented as a hash table in our current implementation. Note that we do *not* store the input reads. Even though the number of k-mers stored in $\hat{G}$ is an order of magnitude smaller than N, the actual memory consumed in bytes by $\hat{G}$ still is 6-7$\times$ larger than N. This is due to auxiliary information per k-mer, used in our current implementation. Note that we use a compact bit representation to represent each k-mer in $\hat{G}$.

4.3 Evaluation of Trade-offs

Parametric study: Table 5 depicts a detailed parametric analysis conducted using the *E_coli* dataset, by varying three factors: 1) k, 2) the CM sketch threshold (τ), and 3) the hash table length for $\hat{G}$. We experimented with values of 32 and 44 for k, 2M and 14M for the hash table length, and varied τ from 200 to 1000. As expected, the number of k-mers, across all the three sets of experiments, continues to diminish with the increase in τ, accompanied with a noticeable decrease in memory consumption as well. However, an excessive increase in τ contributes to lowering the output quality (more particularly, N50). The effect of τ is more pronounced in experiments with higher k-mer lengths, as seen in the case of k=44. This is because a larger value of k produces more distinct k-mers but with reduced frequency of occurrence. Therefore, lowering τ with increase in k is recommended. Results for k=32, however do not show a distinguishable difference in overall quality, with changes in *either* threshold or hash table length, maintaining a steady coverage of over 98%. The hash table length however impacts performance, with a larger length leading to a runtime reduction. Memory consumption, on the other hand, is still dominated by the number of k-mers inserted into $\hat{G}$ (hash table) and therefore, is affected by τ.

Comparing γ vs. β Heuristics: Table 4 presents the results for the two variants viz. *FastEtch*-γ and *FastEtch*-β. These experiments were tested for a higher value of τ, anticipating to expose tradeoffs between better filtering efficacy by the CM sketch vs. output quality. Results in Table 4 show a trade-off between reducing the number of k-mers stored in $\hat{G}$ and preserving the assembly quality (i.e. space vs. quality). Given that *FastEtch-baseline* for τ=400, produced an output of similar quality with approximately 56 million k-mers in $\hat{G}$, we were able to consequently obtain a 25% reduction in space consumption, by producing the same output quality with approximately 42 million k-mers. We also note that the performance of the γ and β variants are comparable, where although it is preferable to favor a larger γ vs. a smaller β, the resultant assemblies derived from both were essentially similar, with the β variant contributing to a slightly higher N50.

4.4 Comparative analysis

We compared *FastEtch* with four other state-of-the-art assemblers — viz. SOAPdenovo, Velvet, ABySS, and Minia. All experiments were executed on a node with 32 cores. Memory consumption corresponds to peak memory usage. However, note that Minia also uses secondary storage. For SOAPdenovo and Velvet, the graph construction and contig generation processes needed to be run separately. For comparison purpose, we use the *E_coli* genome, which was the smallest input for which all the programs ran successfully under at least one comparable parameter setting.

Table 6 shows the results of our comparison. We observed that all three variants of *FastEtch* performs consistently the best in terms of runtime performance. With respect to memory usage, *FastEtch* was second only to Minia, which also uses the disk (disk space not reported). More specifically, *FastEtch* performed between 11.6% to 30.6% faster than the second fastest algorithm, across all the experiments. Also, on an average we consume 20-30% less memory in comparison to the traditional de Bruijn graph based assemblers. Consequently, our γ and β variants were able to accomplish an additional 15% reduction in terms both memory and time, by further reducing the size of $\hat{G}$, without compromising (and in some cases improving) the assembly quality.

In terms of assembly accuracy, *FastEtch* surpasses Velvet and SOAPdenovo by a large margin both in terms of N50 and % genome covered. ABySS and Minia, on the other hand, perform better in terms of N50 and unaligned contig length, albeit taking longer to complete. Note that ABySS is a full-blown assembler with all conventional steps such as (exact) graph construction, error correction and contig generation. Also, ABySS is a parallel code that uses Message Passing Interface (MPI) for parallelization. Minia, on the other hand, performs comparable in quality to ABySS despite using Bloom filters. It is to be noted that *FastEtch* generally produces the best genome coverage, and performs similar in other quality metrics compared to Minia and ABySS, except mainly for the N50 contig length, which is shorter.

Table 5: Parametric study of *FastEtch-baseline* for assemblies of *E_coli* (Cov=80x) across varying factors.

k-mer length k=32								
τ	**Hash table length**	**#k-mers in G**	**N50 (bp)**	**%Genome covered**	**largest alignment (bp)**	**Unaligned length (bp)**	**Time (in sec)**	**Memory (in GB)**
200	2,097,152	58,992,931	14,517	98.19	69,459	843	110.30	2.67
300	2,097,152	57,519,737	13,589	98.19	76,028	596	81.24	2.55
400	2,097,152	56,046,239	13,132	98.13	76,028	1,583	80.80	2.45
600	2,097,152	53,088,519	12,544	98.06	76,028	827	80.01	2.32
800	2,097,152	50,126,785	12,544	98.12	70,151	5,090	77.72	2.20
1000	2,097,152	47,161,170	12,050	98.11	64,181	690	78.11	2.01
k-mer length k=32								
τ	**Hash table length**	**#k-mers in G**	**N50 (bp)**	**%Genome covered**	**largest alignment (bp)**	**Unaligned length (bp)**	**Time (in sec)**	**Memory (in GB)**
200	14,680,064	58,993,225	14,866	98.09	74,408	3,072	73.91	2.74
300	14,680,064	57,522,726	14,490	98.09	79,221	2,898	73.50	2.52
400	14,680,064	56,045,485	14,866	98.15	74,408	3,494	69.62	2.44
600	14,680,064	53,087,350	12,484	98.15	74,408	285	70.80	2.31
800	14,680,064	50,123,837	13,019	97.99	59,493	1,155	73.11	2.16
1000	14,680,064	47,157,443	11,831	98.10	64,214	1,048	68.18	2.04
k-mer length k=44								
τ	**Hash table length**	**#k-mers in G**	**N50 (bp)**	**%Genome covered**	**largest alignment (bp)**	**Unaligned length (bp)**	**Time (in sec)**	**Memory (in GB)**
200	14,680,064	63,109,420	23,992	98.34	110,001	2,476	60.30	2.91
300	14,680,064	61,157,424	19,994	98.30	98,375	14	58.12	2.73
400	14,680,064	59,206,449	16,995	98.27	98,375	1,196	59.51	2.50
600	14,680,064	55,294,516	10,551	98.11	97,334	461	60.63	2.36
800	14,680,064	51,388,555	5,236	97.59	46,178	3,340	61.41	2.18
1000	14,680,064	47,456,212	2,799	96.23	19,483	315	60.50	2.09

Table 6: Comparative evaluation of assemblies generated by *FastEtch* and four other state-of-the-art assemblers. "NA" indicates that the assembler is unable to produce an output.

E_coli (Cov=80x)		**Memory (GB)**	**Time (sec)**	**N50 (bp)**	**%Genome covered**	**Largest alignment (bp)**	**Unaligned contig len. (bp)**
k=32	**FastEtch-baseline (τ=400)**	2.24	70.86	14,866	98.15	83,580	3,494
	FastEtch-γ (τ=1300, γ=900)	1.97	65.67	14,029	98.12	63,544	2,629
	FastEtch-β (τ=1300, β=2)	1.92	62.76	14,155	98.15	78,433	3,792
	SOAPdenovo	8.93	102.22	278	86.15	1,720	375,826
	ABySS	4.82	248.34	22,904	97.76	127,978	0
	Velvet	3.04	126.17	887	95.52	4,803	23,448
	Minia	1.79	80.25	21,183	96.97	127,978	0
k=44	**FastEtch-baseline (τ=200)**	2.43	62.04	23,992	98.34	110,001	2,476
	FastEtch-γ (τ=500, γ=900)	2.22	56.05	24,154	98.34	119,578	165
	FastEtch-β (τ=500, β=2)	2.21	57.35	25,744	98.35	118,836	153
	SOAPdenovo	9.20	90.81	1,174	96.94	7,346	36,548
	ABySS	5.21	250.20	46,085	98.14	166,040	0
	Velvet	NA	NA	NA	NA	NA	NA
	Minia	2.15	89.43	46,092	98.06	166,043	100
Yeast (Cov=100x)		**Memory (GB)**	**Time (sec)**	**N50 (bp)**	**%Genome covered**	**Largest alignment (bp)**	**Unaligned contig len. (bp)**
k=32	**FastEtch-baseline (τ=800)**	7.42	230.56	17,223	94.75	98,269	11,290
	FastEtch-γ (τ=1600, γ=1000)	6.27	219.15	16,561	95.42	103,076	21,223
	FastEtch-β (τ=2400, β=2)	6.52	221.24	18,161	94.97	99,954	14,862
	SOAPdenovo	14.70	355.04	208	74.92	1,241	1,121,246
	ABySS	10.23	810.11	22,371	94.22	107,976	0
	Velvet	12.25	500.25	631	90.19	4,318	76,821
	Minia	2.61	292.17	22,371	94.12	107,976	238
k=44	**FastEtch-baseline (τ=400)**	7.86	220.54	22,612	95.31	131,423	5,750
	FastEtch-γ (τ=800, γ=1000)	7.14	210.73	21,565	95.81	133,711	17,410
	FastEtch-β (τ=1800, β=2)	6.91	191.99	23,335	95.34	131,609	12,359
	SOAPdenovo	12.72	222.49	763	92.51	5,689	150,673
	ABySS	11.40	850.43	33,681	94.75	122,861	0
	Velvet	NA	NA	NA	NA	NA	NA
	Minia	3.58	302.48	34,083	94.68	122,865	717

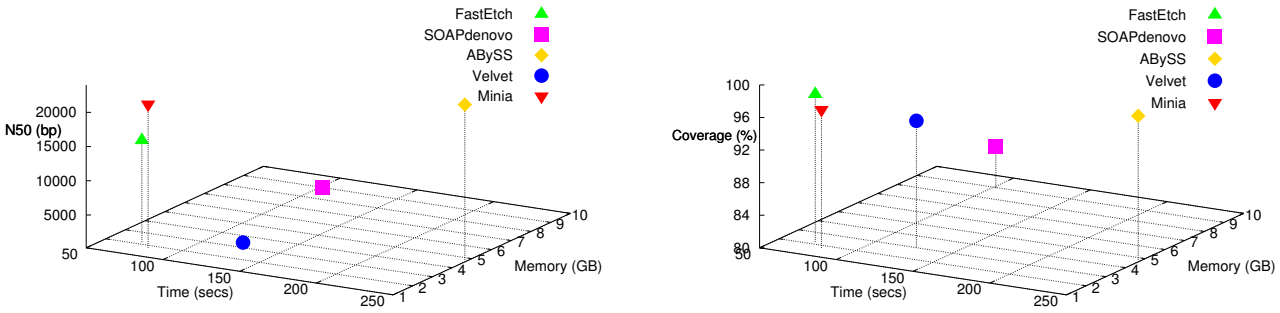


Figure 7: 3-D plot depicting the performance (time, memory, quality) of all assemblers tested, on the *E_coli* dataset (Cov=80x) with k=32. For *FastEtch*, the data plotted corresponds to the *FastEtch-baseline* variant.

This suggests that the current contig extension heuristic we use can be further improved. Fig. 7 summarizes the quality-time-memory comparisons for all the methods tested, using a 3-D representation for the *E_coli* dataset (k=32). Although our approach relies on approximation, we are able to deliver longer contigs and high assembly accuracy, in less time and memory compared to most other short-read assemblers.

5. CONCLUSIONS AND FUTURE WORK

In this paper we introduced *FastEtch*, a novel approach to assemble genomes using sketching. Our approach computes the assembly based on an approximate, space-efficient version of the de Bruijn graph. To the best of our knowlege, *FastEtch* is the first short-read assembler to use the sublinear streaming data structure, CM sketch. Results have demonstrated that, despite approximation, *FastEtch* can produce highly accurate fast assemblies in reduced memory footprint, compared to most other assemblers. Our algorithm also exposes tradeoffs between time, memory and quality, that can be exploited by users.

Future research directions include (but not limited to): i) exploring alternative, more compact strategies to implement the approximate de Bruijn graph; ii) use of error correction heuristics and incorporation of paired-end information to further improve quality; iii) theoretical and analytical studies to further improve filtering efficacy of CM sketch; iv) parallelization on distributed memory machines for scaling to much larger data sets; and iv) extension to other assembly frameworks such as transcriptome assembly.

6. ACKNOWLEDGMENTS

This research was supported in part by U.S. Department of Energy grant DE-SC-0006516. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

7. REFERENCES

[1] J. Butler, I. MacCallum, M. Kleber, et al. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

[2] M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–330, 2008.

[3] J. A. Chapman, I. Ho, S. Sunkara, et al. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011.

[4] R. Chikhi and G. Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1):1, 2013.

[5] T. C. Conway and A. J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[6] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[7] E. Georganas, A. Buluç, J. Chapman, et al. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *High Performance Computing, Networking, Storage and Analysis, SC14*, pages 437–448, 2014.

[8] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, page btt086, 2013.

[9] W. Huang, L. Li, J. R. Myers, and G. T. Marth. Art: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, 2012.

[10] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[11] R. Li, H. Zhu, J. Ruan, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[12] N. Nagarajan and M. Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, 2013.

[13] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[14] K. Salikhov, G. Sacomoto, and G. Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, 9(1):1, 2014.

[15] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[16] J. T. Simpson, K. Wong, S. D. Jackman, et al. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[17] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.