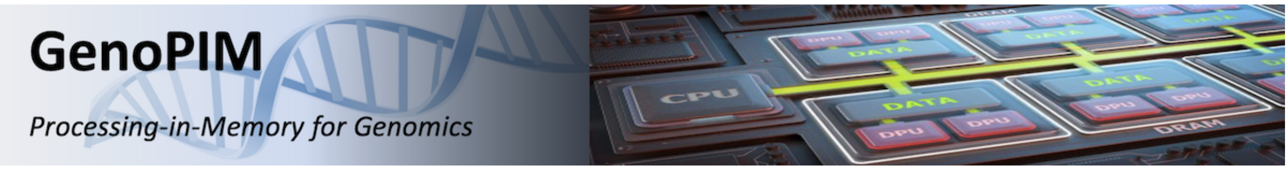


# Characterization of Bloom filters and Graph Algorithms

## Abstract

De Bruijn graphs are a central data structure to process genomic reads. To handle huge volumes of reads more easily and perform computational analyses faster, libraries may rely on efficient compact structures, such as Bloom filters, to build this graph. To improve further these processes, it is critical to understand the shortcomings of current approaches. Therefore, we investigate the performances of Bloom filters and several de Bruijn graph algorithms that use them. In particular, our analysis aims at identifying their memory footprint.

## Table of Contents

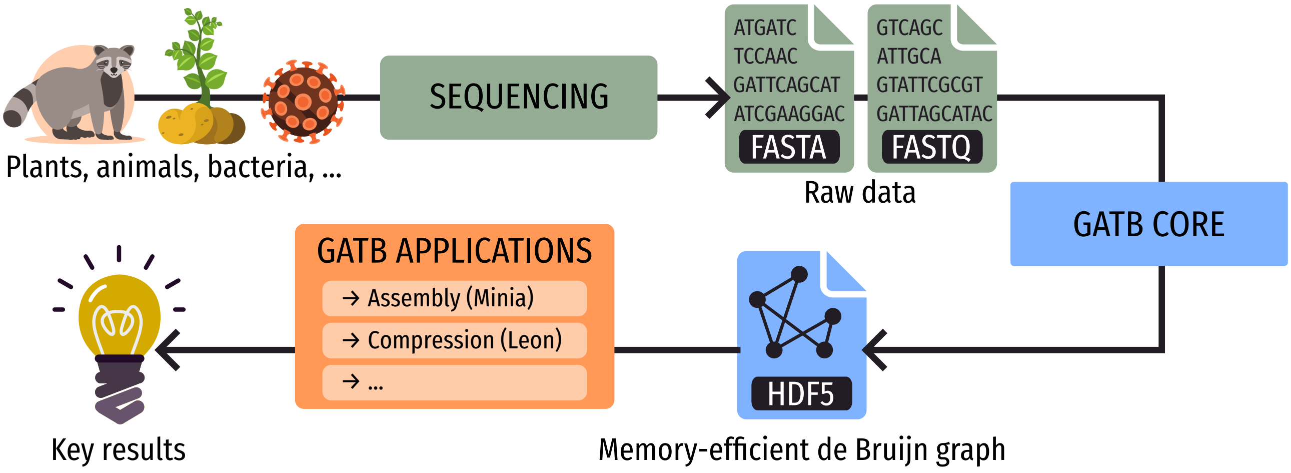


Figure 1: The Genome Analysis Toolbox with de Bruijn graph

## 1 Introduction

De Bruijn graphs (*dBG*) are a central data structure to efficiently process genomic reads. It suits very well the manipulation of the high volume of data generated by HTS technologies since it leverages the sequencing redundancy to compact the data. We consider in this document a node-centric definition of this data structure. Given a set of k-mers $K$ (i.e. substrings of size $k$), the de Bruijn graph of $K$ is a directed graph such that (i) vertices are exactly the elements of $K$ and (ii) an edge connects $u$ and $v$ if the $k-1$ suffix of $u$ equals the $k-1$ prefix of $v$. Given the vertices, edges are thus implicit and we can refer to the graph by its set of k-mers.

If we encode each base pair from $\Sigma = \{A, C, G, T\}$ with 2 bits, then the straightforward way to store a *dBG* requires $2k\,|K|$ bits. Common values for $k$ found in the literature include 23, 25, 27 and 31. For instance, [13] reports 9.35 billion distinct 25-mers in a sequencing dataset from the 1000 Genomes Project [1]. The naive encoding of this graph would thus take $2 \times 25 \times 9.35 \times 10^9$ bits $\sim$ 430 GiB. Although the graph is typically much smaller than the original set of sequencing reads, processing such volumes of data still requires expensive systems with a lot of memory and computational power. As a consequence, much software strives to compact the graph to ease processing and make it accessible to a wider range of systems. We focus on such software in this document.

This document first explains how de Bruijn graphs are built and used in a memory efficient open source library called GATB. Second, we present our analysis setup and methodology. We then review our numerous experimental results. Finally, we draw some conclusions.

## 2 De Bruijn Graphs in the GATB Library

The Genome Analysis Toolbox with de Bruijn graph (GATB) [8] is an open source C++ library developed by the GenScale group from IRISA/CNRS. It provides a set of very efficient tools to analyze NGS datasets and uses compact data structures to run with a very low memory footprint. The library splits the process into two distinct parts, as illustrated in Figure 1:

1. The core module takes as input raw FASTA/FASTQ data and produces a memory efficient representation of the *dBG* stored in a HDF5 file. The library provides two representations:

   (a) The Bloom-based graph (*BdBG*) [7] uses a Bloom filter and an adjacent set structure to hold critical false positives.
   (b) The Unitigs-based graph (*UdBG*) [6] compacts long simple paths using a minimizer hashing technique.

2. Tools take the graph as input and perform various tasks like compression, assembly, read error correction, etc. The library provides a rich API to develop new tools on top of the core module.

In the following subsections, we give more details about the two graph representations and their underlying data structures.

## 2.1 Bloom-based de Bruijn Graphs

Bloom filters [4] are a memory-efficient probabilistic set data structure. They support two operations: (i) inserting an element and (ii) querying the presence of an element. A filter is a vector of bits of size $m$ with all cells initialized to 0. It uses $h$ independent hash functions to hash any element into a list of $h$ indexes. As shown in the basic implementation in Listings 1 and 2, the insertion operation sets the bits at the resulting indexes to 1. To query the filter, we look at the indexes and return a positive if all corresponding cells contain a 1. However, the lookup answer should be interpreted as either *No* or *Maybe Yes*. Hashing collisions can indeed lead one to believe some elements were inserted in the filter when they were not. The false positive probability is given by the following formula, where $n$ is the total number of elements inserted into the filter:

$$\mathbb{P}[\mathrm{FP}] \sim \left(1 - e^{-\frac{hn}{m}}\right)^h$$

On the other hand, false negative are impossible: $\mathbb{P}[\mathrm{FN}] = 0$.

Listing 1: Basic Bloom filter insertion

```
void insert(const ItemType item) {
  for (int i = 0; i < h; i++) {
    bloom[hash(item, i)] = 1;
  }
}
```

Listing 2: Basic Bloom filter lookup

```
bool contains(const ItemType item) {
  for (int i = 0; i < h; i++) {
    if (bloom[hash(item, i)] == 0) { return false; }
  }
  return true;
}
```

The GATB library contains several implementations of Bloom filters:

- *Basic*, as described previously.
- *Cache*, where hash computations are tweaked to have all indexes close to each other and thus improve the cache locality.
- *Neighbor*, which is specific to k-mers and improves the cache locality with the neighboring k-mers in the *dBG*.

All these have variants that incorporate synchronization around the underlying bit vector write operations in order to use the filters in a multithreaded environment.

To provide an exact graph representation despite the probabilistic nature of Bloom filters, the GATB library computes an adjacent structure to keep track of critical false positives (*cFP*). This structure has two implementations:

- *Original*, which uses a set to hold false positive elements.

- *Cascading*, which uses cascading Bloom filters to lower the memory footprint of the *cFP* [16].

Finally, the library provides a mechanism to traverse the graph and mark visited nodes. To this end, it computes the set of complex nodes of the graph, i.e. the nodes that have their in-degree or out-degree strictly greater than 1, and inserts them into a hash table. Other nodes are part of simple paths, and their traversal status is thus implicit, given the marking of the complex nodes.

To summarize, the *BdBG* construction performs the following steps:

Step 1: Count the abundance of all k-mers (DSK) [14].
Step 2: Insert the solid k-mers[1] into a Bloom filter.
Step 3: Build the *cFP*.
Step 4: Compute branching information and store complex nodes in a hash table.

### 2.2 Unitigs-based de Bruijn Graphs

In the de Bruijn graph, a unitig is a simple path that cannot be extended further without introducing some branching. The *UdBG* is represented by two sets: (i) the set of unitigs and (ii) the set of all remaining k-mers that are not part of any unitig. Unitigs are encoded by their sequence, which takes a lot less space than storing all of their composing k-mers.

The algorithm that computes the *UdBG*, called BCALM2 [6], uses a minimizer hashing technique. Given a string $u$, the l-minimizer of $u$ is the smallest l-mer of $u$. BCALM2 distributes k-mers to different buckets depending on the l-minimizer of their $(k-1)$ prefix (left minimizer) and the l-minimizer of their $(k-1)$ suffix (right minimizer). The k-mers whose left and right minimizers are not equal are duplicated and inserted into two buckets. Instead of using a lexicographic order to compute the smallest l-mer, the library orders l-mers by increasing frequency to achieve better bucket load balancing.

Each bucket is then processed in parallel to perform compaction, i.e. merge all pairs $(u, v)$ where $u$ is the only in-neighbor of $v$ and $v$ the only out-neighbor of $u$. Finally, a reunification step is necessary to glue back the duplicated strings. This computation uses a union-find data structure implemented with a minimal perfect hash function.

To summarize, the *UdBG* construction performs the following steps:

Step 1: Count the abundance of all k-mers.
Step 2: Distribute solid k-mers to buckets
Step 3: Compact each bucket.
Step 4: Reunite and glue the duplicated strings.
Step 5: Output the set of unitigs and the set of remaining k-mers.

## 3 Analysis Setup and Methodology

We analyze programs with the VTune Profiler software. It is a top-down analysis tool for x86 architectures [17]. More precisely, we run the micro-architecture analysis that categorizes micro-operations depending on whether they are allocated, retired, or stalled, as illustrated in Figure 2. It indicates red flagged metrics and allows to spot the pipeline stalls.

[1] We call solid the k-mers that appear more than a given threshold, usually 2 or 3. The goal of this process is to eliminate low-abundance k-mers that usually result from sequencing errors.

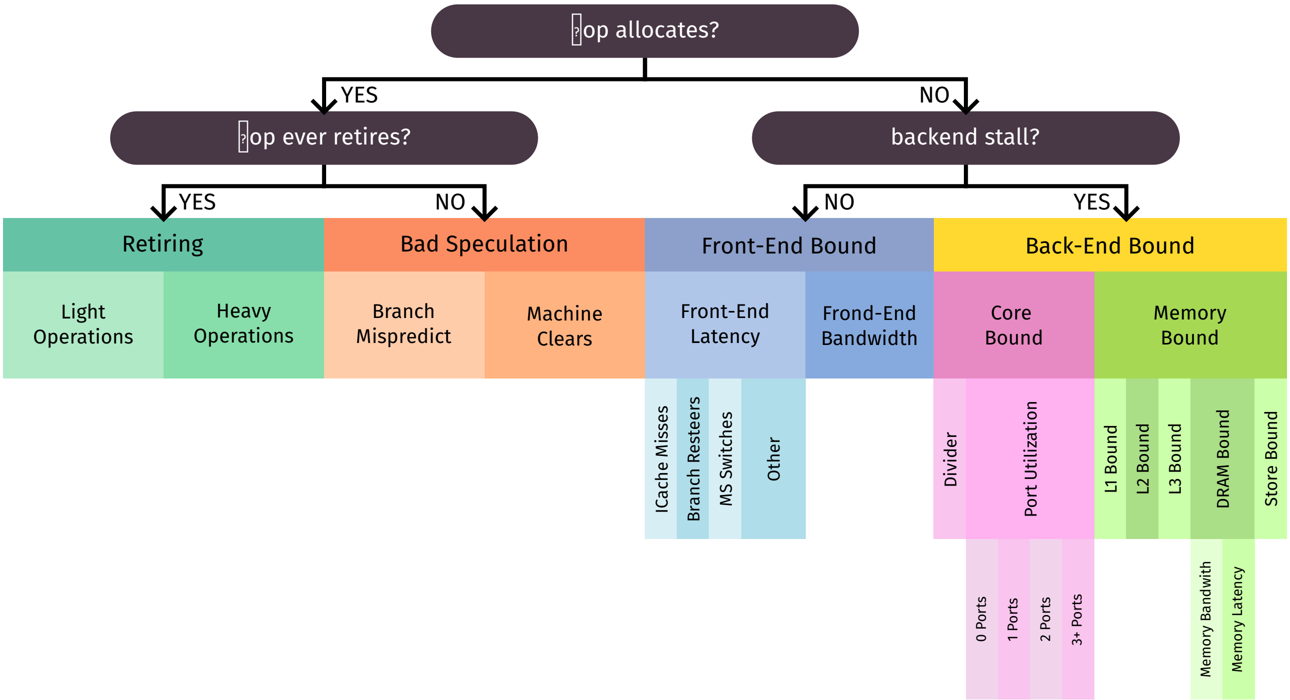


Figure 2: Top-down analysis with VTune Profiler

To present results, we plot the distribution of the following categories in terms of pipeline slot percentage: Retiring, Bad Speculation, Front-End Bound, Core-Bound and Memory Bound. We display the two sub-categories of the Back-End Bound metric instead of the top level as they are particularly relevant to our analysis. Moreover, we report the runtime in seconds. The ideal case consists of an application with 100% Retiring as it exhibits no particular stalls and maximizes the number of instructions per cycle.[2]

Finally, we consider the red flag thresholds given by VTune. For instance, the software considers that having less than 15% of Bad Speculation is acceptable and suggests investigating only if it exceeds this value. We thus highlight on our plots the zones where metrics exceed their corresponding red flag thresholds.

For some experiments, we instrumented the code with the Intel® Instrumentation and Tracing Technology (ITT) API to better control the collection of data by VTune. As shown in Listing 3, we can select precisely the part of the code that we analyze.

Listing 3: Instrumenting a benchmark

```
#include <ittnotify.h>

__itt_resume();
// Code to analyze
__itt_pause();
```

We describe in Table 1 the system setup we use for our experiments. Cache sizes are particularly important in our context because we wish to analyze what happens when the data structures we manipulate become too big to fit into the different caches.

[2] However, it should be noted that it does not mean the application cannot be optimized further. For instance, using vectorized instructions where applicable may improve performance.

| | |
|---|---|
| **Hardware** | |
| Processor | Intel® Core$^{\text{TM}}$ i7-8650U 4 cores @ 4.2 GHz. |
| L1 Cache | 256 KiB |
| L2 Cache | 1 MiB |
| L3 Cache | 8 MiB |
| Memory | 32 GiB DDR4 @ 2.4 GHz |
| **Software** | |
| OS | Fedora Workstation 37 |
| VTune Profiler | Version 2023.1.0 |

Table 1: System configuration

# 4 Bloom Filters Micro-benchmarks

Since Bloom filters are a core component of the *BdBG*, we decided to first perform some micro-benchmarks to analyze the memory footprint of their two elementary operations. In this section, we start by describing our micro-benchmarks and the different scenarios they simulate. Then we review our analysis results in both single-threaded and multithreaded.

## 4.1 Scenarios

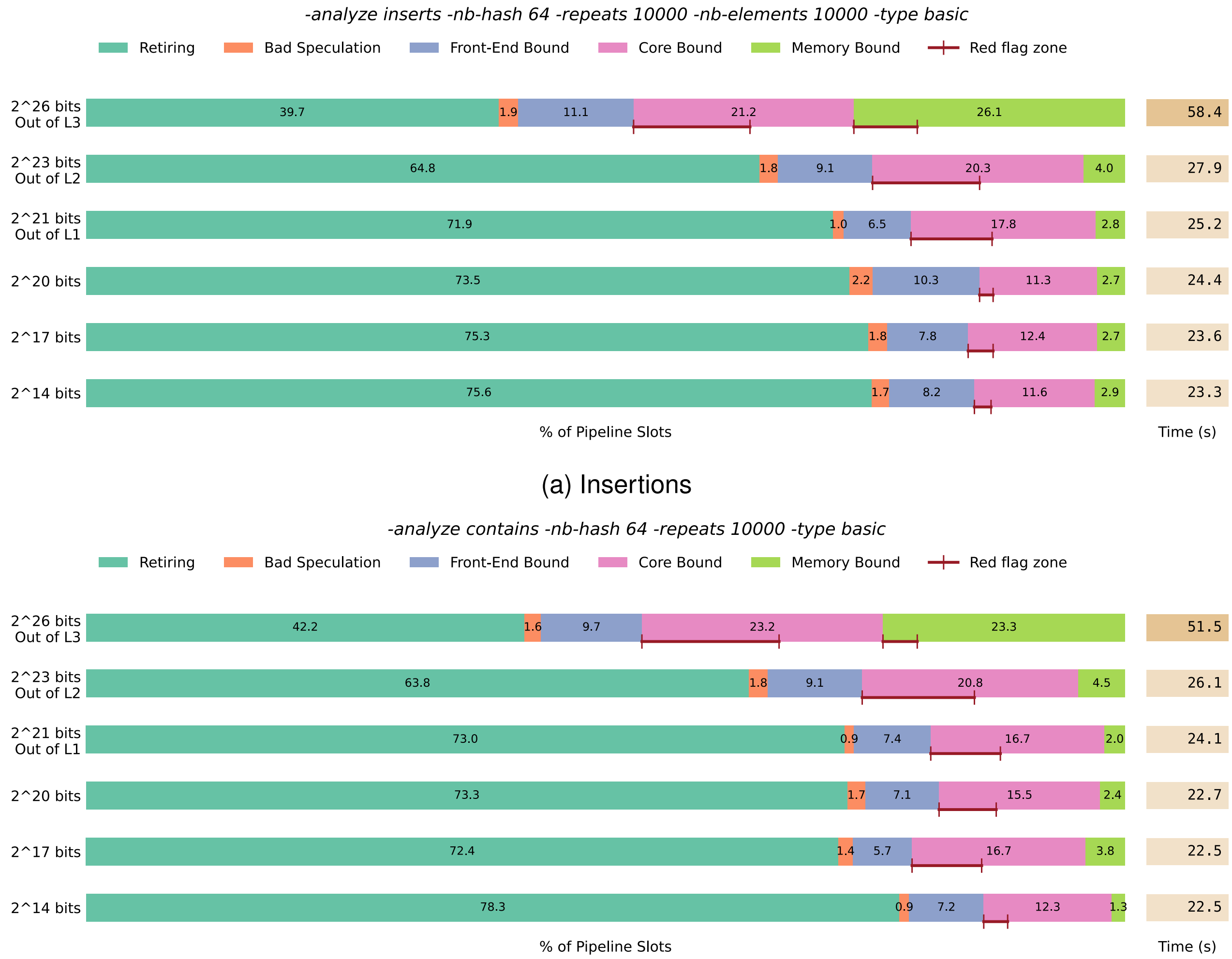
We consider a first scenario where we wish to insert many elements into a Bloom filter and do not plan to query the filter in-between. We can thus insert the elements in any order. The code for this micro-benchmark is given in Listing 4. We use OpenMP to execute the insertions in parallel with 8 threads. We repeat the operation many times to increase the runtime and allow VTune to collect enough data. This scenario typically corresponds to step 2 of the *BdBG* construction in the GATB library, where it inserts all solid k-mers into a filter.

Listing 4: Insertions micro-benchmark

```
BloomFilter* bloom_filter = new_bloom(type, size, nb_hash);
for (int repeats = 0; repeats < nb_repeats; repeats++) {
  #pragma omp parallel for num_threads(8) if(do_parallel)
  for (int i = 0; i < nb_elements; i++) {
    bloom_filter->insert(i);
  }
}
```

Then, we consider a second scenario, where we wish to query a Bloom filter that is not modified between lookups. We can thus execute the queries in any order. The code of this micro-benchmark is given in Listing 5. We simulate here a worst-case scenario by setting all the filter bits to 1, which means all hashes will be computed, and all cells will be queried[3]. Also, we use the `volatile` keyword to prevent compiler optimizations that would strip away most lookup calls and defeat the purpose of this micro-benchmark. This scenario typically corresponds to step 3 of the *BdBG* construction in the GATB library, where it queries the filter a lot to compute the *cFP*.

[3] In a realistic setting, we would want a low false positive probability, so we would configure the filter to end up with many 0 cells, and most queries would return very quickly.



(a) Insertions

(b) Lookups

Figure 3: Varying the size of the basic Bloom filter (single-threaded)

Listing 5: Lookups micro-benchmark

```
BloomFilter* bloom_filter = new_bloom(type, size, nb_hash);
bloom_filter->fill1(); // Worst case scenario
volatile bool result = false; // Prevent compiler optimization
for (int repeats = 0; repeats < nb_repeats; repeats++) {
  #pragma omp parallel for num_threads(8) if(do_parallel)
  for (int repeats2 = 0; repeats2 < nb_repeats; repeats2++) {
    result |= bloom_filter->contains(repeats2);
  }
}
```

## 4.2 Results in Single-threaded

First, we vary the size of the basic Bloom filter in Figure 3. For both insertions and lookups, we notice a huge difference in runtime as soon as the filter does not fit into the L3 cache. VTune clearly flags this case as Memory Bound, and more precisely as DRAM Bound. This behavior was quite expected since the use of hash functions induce a lot of random accesses and do not rely on memory locality.

Next, we compare the basic filter with the cache-friendly implementation of the GATB library (cf.

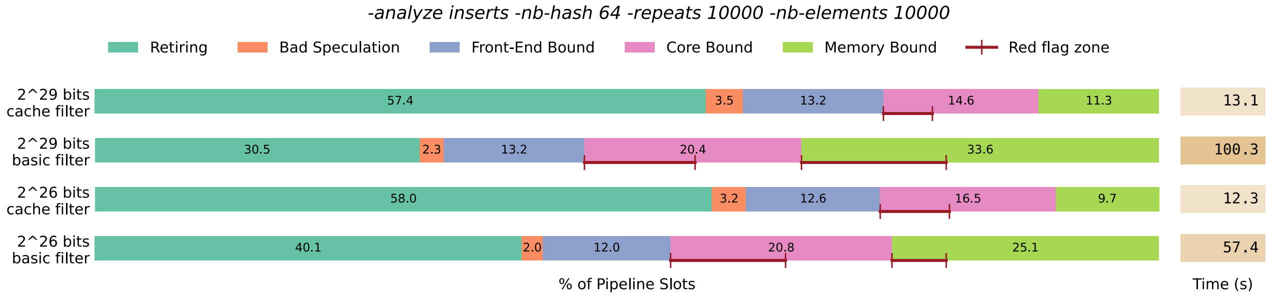


Figure 4: Comparing basic and cache Bloom filters when out of L3

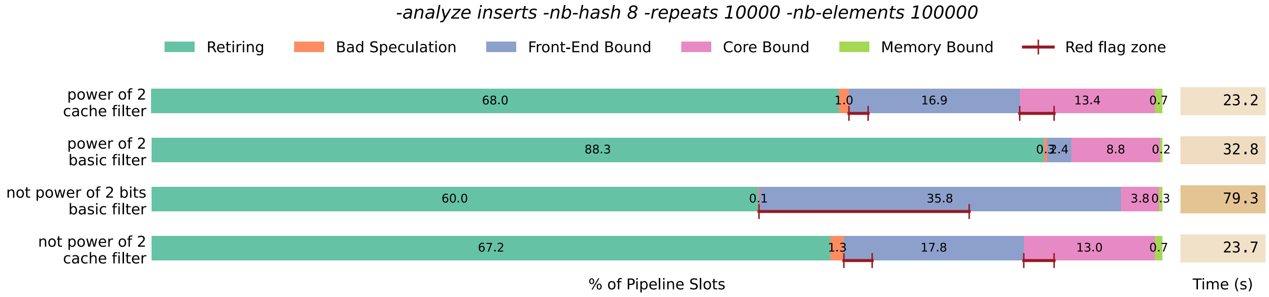


Figure 5: Impact of the Bloom filter size being a power of 2 ($2^{16}$ bits) or not ($2^{16} - 1$ bits)

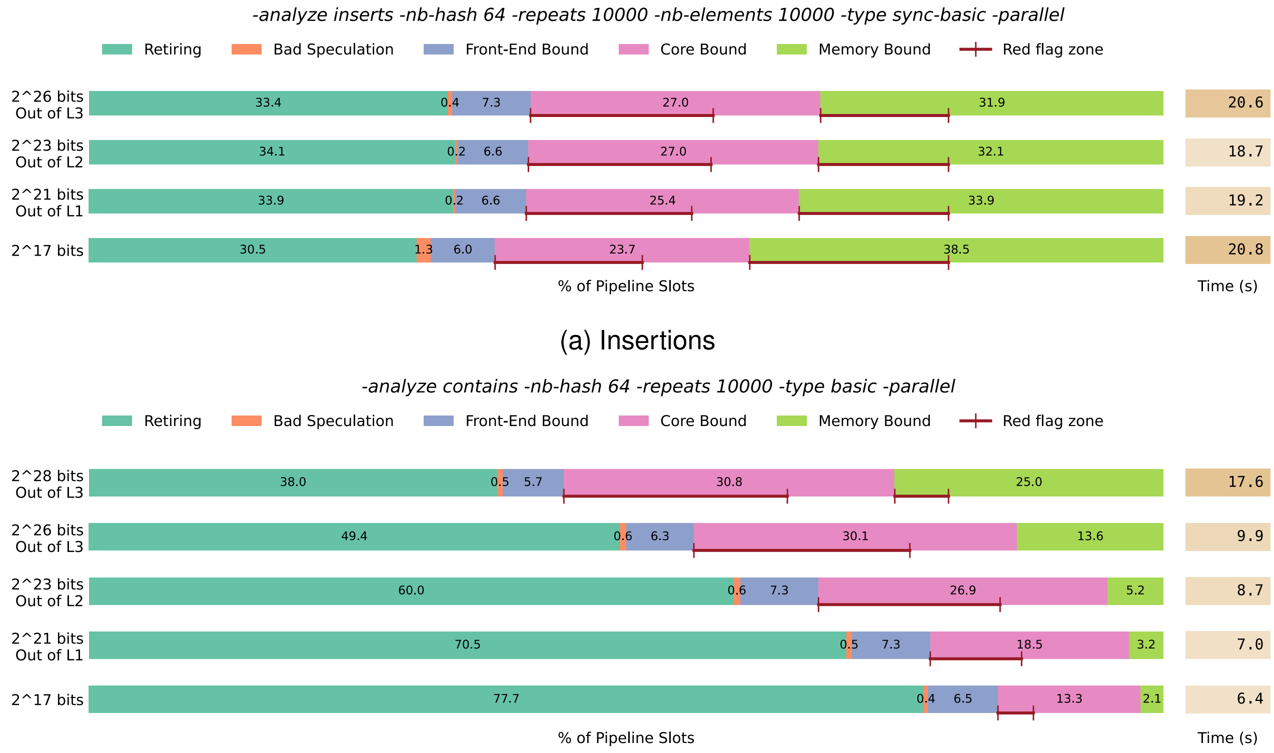
Section 2.1) in Figure 4. We see that the cache filter is efficient at mitigating the issue mentioned previously and is less Memory Bound. The percentage is however still growing with the filter size, and we expect this to become a problem with real-world applications that use huge datasets.

We also run some experiments with filter sizes that are not a power of 2 in Figure 5. We notice the basic implementation shows a significant difference in runtime and that the application becomes Front-End Bound when the size is not a power of 2. In this last case, some optimizations are indeed not possible, and we have to rely on modulo computations instead of bitwise operators. VTune shows that this causes a lot of Microcode Sequencer Switches that harm the performance. We thus recommend using the basic filter with its size being a power of 2 as much as possible. However, the cache filter implementation does not suffer from the same issue as it relies only once on a modulo for the first hash computation and use simpler operators for the next as part of its process to improve the memory locality.

## 4.3 Results in Multithreaded

We vary again the filter size and run the application with 8 threads using OpenMP. Regarding the lookups, we notice in Figure 6 a similar behavior to before. However, the insertions are now all Memory Bound no matter the filter size. More precisely, VTune reports high values of Lock Latency. This happens because of the synchronization that is required between threads when modifying the bit vector. This is not specific to the basic implementation. Additional experiments show that the cache filter exhibits the same Memory Bound behavior for insertions.

One way to deal with big Bloom filters, especially in the context of parallel architectures or distributed environments, is partitioning the filter or using several smaller filters. We consider the latter for the following experiment. We first perform insertions in parallel in one big filter of size $2^{29}$ bits with the



(a) Insertions

(b) Lookups

Figure 6: Varying the size of the basic Bloom filter (multithreaded)

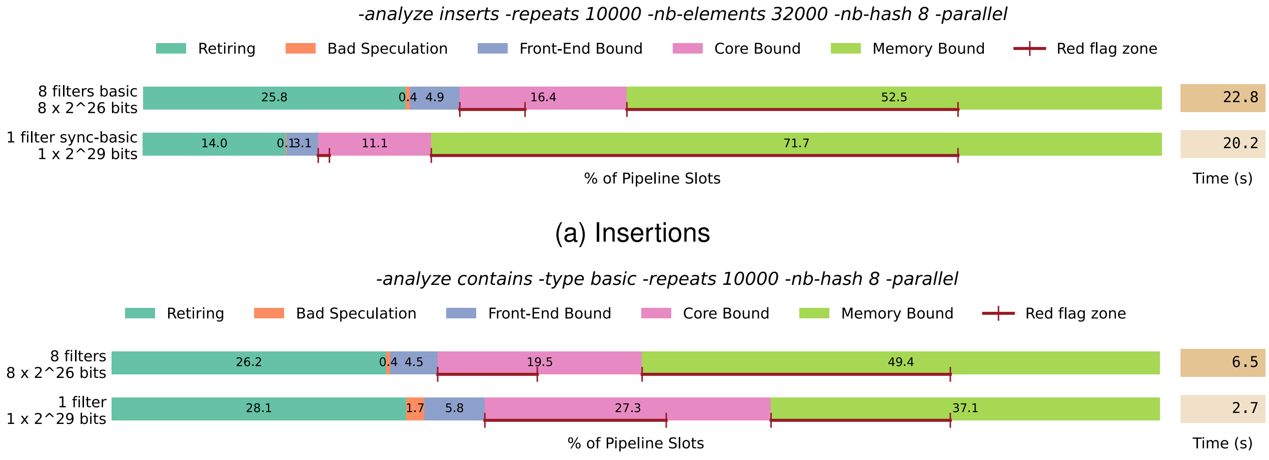
synchronized basic implementation. Then we give each thread its own smaller filter of size $2^{26}$ bits to fill in total the same memory space with all 8 threads. Note that we can use the basic implementation with no synchronization since each filter is modified by only one thread. We then repeat the experiment for the lookups. We see in Figure 7 that having one big filter is more Memory Bound regarding insertions because of the synchronization process but still performs faster. Regarding lookups, VTune red flags the Memory Latency metric for both cases but also Memory Bandwidth for the smaller filters, which explains why the latter perform worse. Despite having their own filter, the fact that threads share the same main memory remains an important performance bottleneck.

## 5 GATB Benchmarks

We now run benchmarks using the GATB library. To this end, we use several read datasets, both synthetically generated and from real-world sequencing data. Their different characteristics are given in Table 2. To produce synthetic datasets, we generate a long sequence of base pairs and then pick random reads with 10% overall overlap. We obtained two synthetic datasets:

- synth1 is error-free.
- synth1-err0.01 mimics some sequencing errors: each base pair in each read has a 0.01% chance to produce either a deletion, substitution, or insertion.

Real-world sequencing data were downloaded from the European Nucleotide Archive [11]. All our experiments in this section are run in parallel with 8 threads.



(a) Insertions

(b) Lookups

Figure 7: One big filter or 8 smaller filters in multithreaded

| Identifier | Organism | Read size | Read count | Base count | Fasta size |
|---|---|---|---|---|---|
| **Real-world** | | | | | |
| SRR857303 | *E. coli* | 195 | 2,581,532 | 495.4 Mbp | 598 MiB* |
| SRR959239_1 | *E. coli* | 98 | 2,686,416 | 263.3 Mbp | 437 MiB† |
| SRR1870605_1 | *E. coli* | 242 | 1,119,218 | 271.3 Mbp | 304 MiB§ |
| SRR1870605_2 | *E. coli* | 242 | 1,119,218 | 272.9 Mbp | 304 MiB§ |
| SRR065390_1 | *C. elegans* | 100 | 33,808,546 | 3.38 Gbp | 5.3 GiB¶ |
| **Synthetic** | | | | | |
| synth1 | N/A | 80 | 10,000,000 | 800 Mbp | 899 MiB |
| synth1-err0.01 | N/A | 80 | 10,000,000 | 800 Mbp | 899 MiB |

*https://www.ebi.ac.uk/ena/browser/view/SRR857303 †https://www.ebi.ac.uk/ena/browser/view/SRR959239
§https://www.ebi.ac.uk/ena/browser/view/SRR1870605 ¶https://www.ebi.ac.uk/ena/browser/view/SRR065390

Table 2: Read datasets description

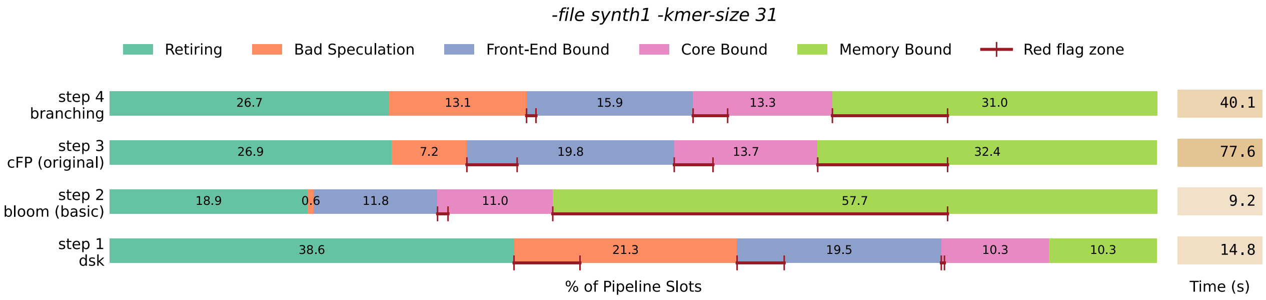


Figure 8: *BdBG* construction with the basic filter

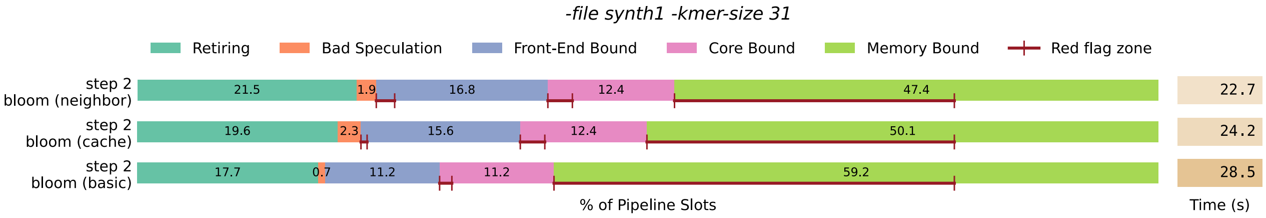


Figure 9: Step 2 of *BdBG* construction with different filter types

## 5.1 Bloom-based Graph Construction

We start with a complete graph construction in Figure 8 where we use the basic filter and the original *cFP*. We notice the following:

- DSK has significant Bad Speculation and is Front-End Bound because of branch mispredictions.
- Step 2 is Memory Bound as expected since it performs many Bloom filter insertions.
- Steps 3 and 4 are also Memory Bound because they query the Bloom filter a lot.

These results are consistent with our micro-benchmarks observations from Section 4.

Next, we focus on Step 2 and vary the filter type. Because we input read datasets and thus k-mers (contrary to the micro-benchmarks where elements were integers), we can now use the neighbor filter implementation from GATB. We see in Figure 9 that all cases are significantly Memory Bound because of the Lock Latency metric. In this setting, we are inserting approximately 79 M solid k-mers into a filter of size around 857 MiB. The cache and neighbor filters perform better than the basic implementation, but their locality improvements do not eliminate all issues when used with such huge amounts of data.

We run the whole construction again with the neighbor filter implementation. We notice in Figure 10 that steps 3 and 4 are not Memory Bound. This filter implementation significantly improves performance when it comes to lookups.

We now focus on step 3 and consider the three different stages of the *cFP* construction algorithm from [7]:

Step 3.1: Compute the set of neighbor extensions that give false positives. This queries the Bloom filter built at Step 2.

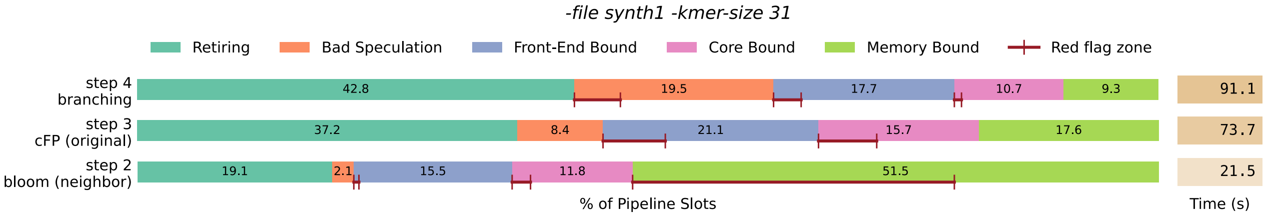


Figure 10: *BdBG* construction with the neighbor filter

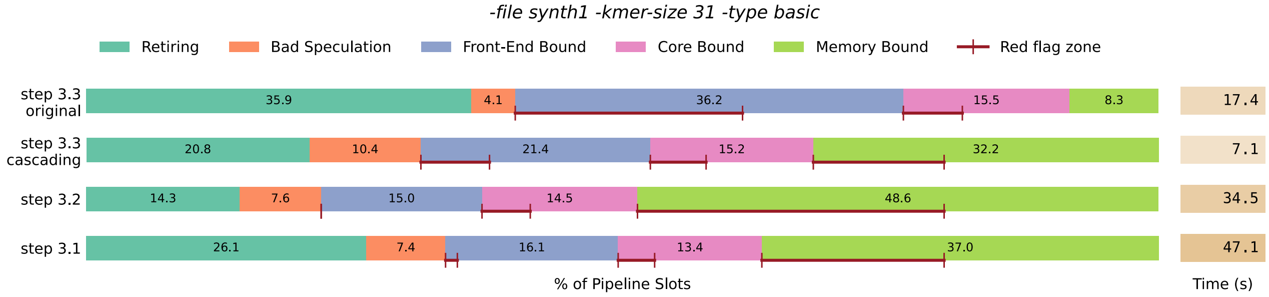


Figure 11: *cFP* construction with the basic filter

Step 3.2: Filter this set to keep only the elements that appear in the dataset. This queries the k-mers bank.
Step 3.3: Store the result either in a set (*original*) or in cascading bloom filters (*cascading*).

We see in Figure 11 that the steps 3.1 and 3.3 (*cascading*) that involve the basic Bloom filter are Memory Bound.

## 5.2 Bloom-based Graph Applications

We now consider that the *BdBG* is constructed and stored in a HDF5 file. We wish to analyze applications built on top of the GATB core module that use this file to perform various genomic tasks.

### *5.2.1 Compression*

We run the tool Leon [3] that efficiently compresses raw read datasets. It uses the neighbor filter implementation. We experiment in Figure 12 with several read datasets. We notice similar results for all cases, with significant Bad Speculation and Front-End Bound due to branch mispredictions. Contrary to the construction stage, applications look at the actual data more precisely and are more likely to use conditionals to perform complex tasks. This kind of performance bottleneck is thus to be expected.

We distinguish several stages in the main compression function of Leon:

Step 1: Build k-mers.
Step 2: Search for an existing anchor.
Step 3: If not found, create and insert an anchor. This step queries the Bloom filter.
Step 4: Encode the read.

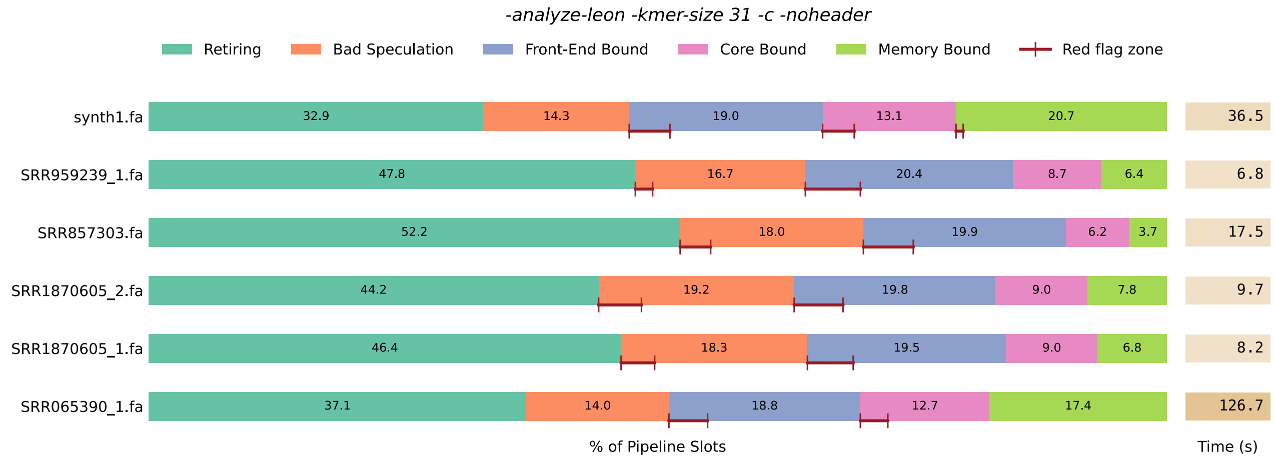


Figure 12: Compressing several datasets with Leon

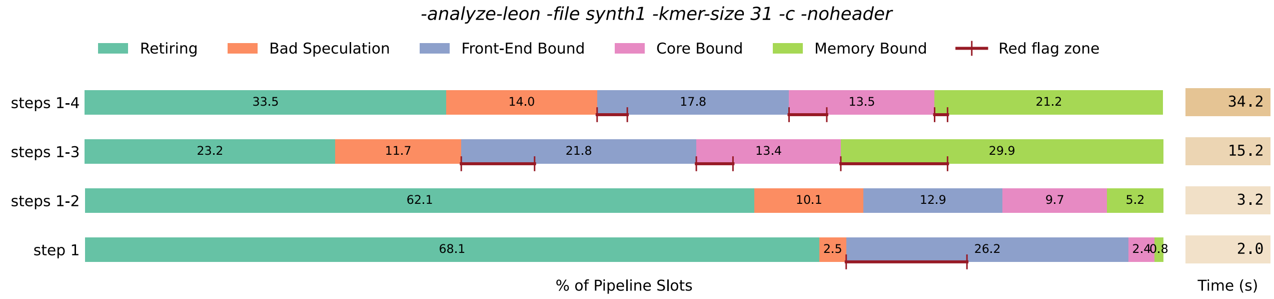


Figure 13: Leon steps

Since these operations are run in a function that is dispatched and executed in parallel on every sequence, we cannot easily instrument the code to focus the data collection. As a workaround, we add artificial `return` statements and execute the program one step further each time. We can thus observe the results of each step in comparison with the previous one. We see in Figure 13 that Step 2 adds Bad Speculation and that Step 3 introduces Memory Bound issues. This seems consistent: Step 2 heavily depends on the actual k-mers data and Step 3 performs lookups on the filter. We also notice that Step 4 (i.e. the encoding) is the longest and hides some of the bottlenecks from the previous steps. However, Step 3 still has a significant runtime contribution overall and could benefit from some better performance regarding its memory footprint.

### 5.2.2 Read error correction

We consider Bloocoo [2] which is a k-mer spectrum-based read error corrector and relies on the *BdBG*. We experiment with the cache implementation. In Figure 14, we notice similar results for all datasets except synth1 with significant Bad Speculation and Front-End Bound bottlenecks. The synthetic dataset synth1 appears to be the most Memory Bound and Bloocoo logically finds no error to correct while it performs significant amounts of corrections in all others, including synth1-err0.01 (e.g. around 8.7 million for SRR065390_1).

We conclude that the performance depends on the actual data and that the process would be Memory Bound for all cases if the pipeline wasn't stalled in its front stage because of branch mispredictions. Additional experiments show similar results with the neighbor filter implementation.

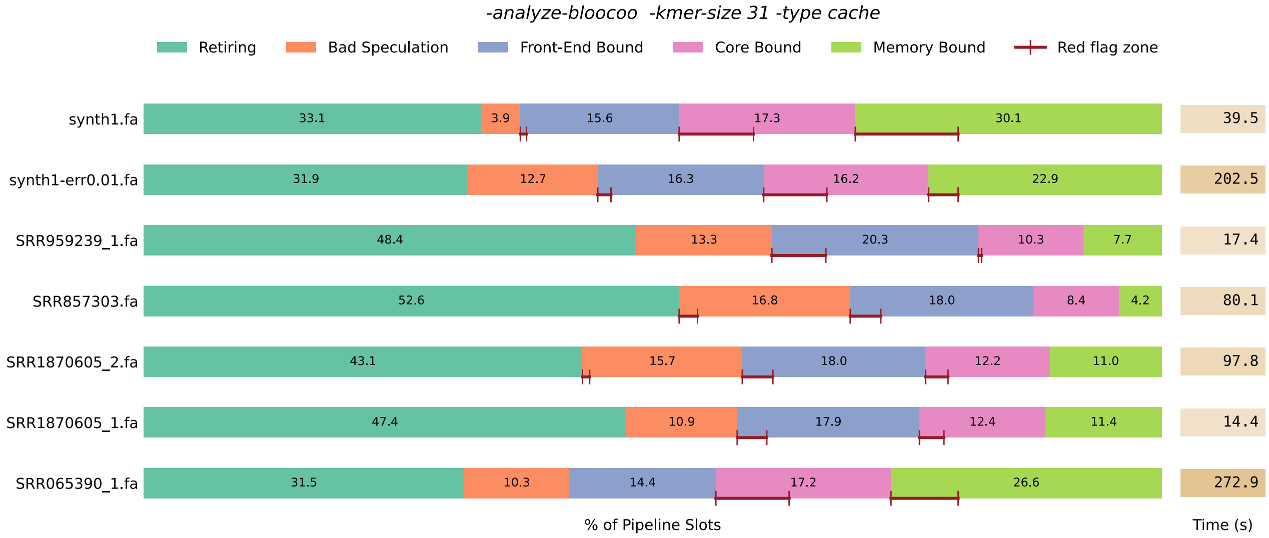


Figure 14: Correcting several datasets with Bloocoo

## 6 Conclusions

In this document, we considered de Bruijn graph algorithms and analyzed the performance of some software. In particular, we ran extensive experiments with the GATB open source library and focused on Bloom filters, as they are a very efficient and compact data structure. We noticed the latter are fundamentally memory bound because of the hash computations that cause the program to access randomly different parts of the memory. As a consequence, all algorithms that rely on Bloom filters and perform many insertions and lookups suffer from this bottleneck. Studies like [9] have shown PIM architectures are especially suited for memory bound workloads. We thus believe implementing Bloom filters on these architectures could be promising performance-wise for manipulating de Bruijn graphs and running various genomic tasks.

## References

[1] 1000 Genomes Project Consortium, Gonçalo R. Abecasis, David Altshuler, Adam Auton, Lisa D. Brooks, Richard M. Durbin, Richard A. Gibbs, Matt E. Hurles, and Gil A. McVean. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, October 2010.

[2] Gaetan Benoit, Dominique Lavenier, Claire Lemaitre, and Guillaume Rizk. Bloocoo, a memory efficient read corrector. September 2014.

[3] Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinformatics*, 16(1):288, September 2015.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] Nicolas Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal RNA-Seq quantification, May 2015. arXiv:1505.02710 [cs, q-bio].

[6] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, June 2016.

[7] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):22, September 2013.

[8] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. GATB: Genome Assembly & Analysis Tool Box. *Bioinformatics*, 30(20):2959–2961, October 2014.

[9] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware. pages 1–7. IEEE Computer Society, October 2021.

[10] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, September 2017.

[11] Rasko Leinonen, Ruth Akhtar, Ewan Birney, Lawrence Bower, Ana Cerdeno-Tárraga, Ying Cheng, Iain Cleland, Nadeem Faruque, Neil Goodgame, Richard Gibson, Gemma Hoad, Mikyung Jang, Nima Pakseresht, Sheila Plaister, Rajesh Radhakrishnan, Kethi Reddy, Siamak Sobhany, Petra Ten Hoopen, Robert Vaughan, Vadim Zalunin, and Guy Cochrane. The European Nucleotide Archive. *Nucleic Acids Research*, 39(suppl_1):D28–D31, January 2011.

[12] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, March 2011.

[13] Páll Melsted and Jonathan K. Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, August 2011.

[14] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652, January 2013.

[15] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: Identifying frequent k -mers with cache-efficient algorithms. *Bioinformatics*, 30(14):1950–1957, July 2014.

[16] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology*, 9(1):2, February 2014.

[17] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. pages 35–44, March 2014.