High-performance processing of Next-Generation Sequencing data on CUDA-enabled GPUs

Dissertation
for the attainment of the degree
"Doctor of Natural Sciences"
at the Department of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University
in Mainz

Felix Kallenborn

born in Bad Kreuznach, Germany.
Mainz, January 28, 2024

1. Reviewer	Prof. Dr. Bertil Schmidt
2. Reviewer	Prof. Dr. Andreas Hildebrandt
Date of oral examination	24.01.24

Abstract

With the technological advances in the field of genomics and sequencing, the processing of vast amounts of generated data becomes more and more challenging. Nowadays, software for processing large-scale datasets of sequencing reads may take hours to days to complete, even on high-end workstations. This explains the need for new approaches to achieve faster, high-performance applications. In contrast to traditional CPU-based software, algorithms utilizing the massively-parallel many-core architecture and fast memory of GPUs are potentially able to deliver the desired performance in many fields.

In this thesis, we introduce two novel GPU-accelerated applications, CARE and CAREx, for common steps in sequence processing pipelines, error correction and read extension of Next Generation Sequencing (NGS) Illumina data, to improve the results of down-stream data analysis. To the best of our knowledge, CARE and CAREx are the first modern GPU-accelerated solutions for the respective problems. A key component of our algorithm is the identification of similar DNA sequences within a dataset. For this purpose, we developed a minhashing-based index data structure for large-scale read datasets. In conjunction with our fast bit-parallel shifted hamming distance computations, this allows for the efficient identification of similar reads. The resulting set of similar sequences is subsequently arranged into a gap-free multiple-sequence alignment to solve the problem at hand.

Sequencing machines introduce both systematic errors and random errors. CARE, Context-Aware Read Error corrector, accurately removes errors introduced by NGS sequencing machines during the initial sequencing of a biological sample. With the help of a pre-trained Random Forest, CARE generates two orders-of-magnitude fewer false positives than its competitors. At the same time, it shows similar numbers of true positives.

Read extension describes the process of elongating DNA sequences. The presence of longer sequences improves the resolution of more, larger structures within a genome. CAREx, Context-Aware Read Extender, produces longer sequences, so called pseudo-long reads, by connecting the two reads of read pairs which were sequenced in close proximity. Evaluation shows that CAREx produces significantly more highly accurate pseudo-long reads than the state-of-the-art.

With algorithms tailored towards high-performance GPU computations, both CARE and CAREx run significantly faster than the CPU-based competitors, while, at the same time, produce more accurate results. The processing of a large Human dataset with 30x coverage with CARE requires less than 30 minutes using a single A100 GPU. This time can be further reduced down to 10 minutes on multi-GPU systems. In contrast, CPU-based tools like Musket or BFC take 3 hours and 1.5 hours, respectively. Read extension of a Human dataset with CAREx takes 3.3 hours to complete on a single GPU, whereas Konnector2 requires over a day to complete.

This shows that large-scale sequence processing can greatly benefit from the usage of GPUs, and that multiple-sequence alignment-based algorithms should be considered despite their increased complexity because they provide great accuracy. While our general building blocks have been tailored towards our needs for error correction and read extension, they could also prove useful in other GPU-accelerated applications that process sequence data.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Thesis structure . . . 3
1.3. Publications . . . 3

2. Background **5**
2.1. Bioinformatics . . . 5
2.2. Sequence similarity and sequence alignment . . . 7
2.2.1. Hamming Distance . . . 7
2.2.2. Pair-wise sequence alignment . . . 7
2.2.3. Multiple-sequence alignment . . . 10
2.3. GPU computing . . . 11

3. Related Work **17**

I. Sequencing read error correction **21**

4. CARE: Context-Aware Read Error correction **23**
4.1. Publications . . . 24
4.2. Related Work . . . 25
4.2.1. K-mer-based methods . . . 25
4.2.2. MSA-based methods . . . 26
4.3. Algorithm . . . 26
4.3.1. Construction phase . . . 27
4.3.2. Correction phase . . . 27
4.3.3. Output phase . . . 34
4.4. Implementation . . . 34
4.4.1. Data structures . . . 34
4.4.2. CPU version . . . 38
4.4.3. Single-GPU version . . . 39
4.4.4. Batched minhasher queries . . . 40
4.4.5. Bit-parallel hamming distance . . . 42
4.4.6. MSA construction . . . 43

4.4.7. Random Forest 44
4.4.8. Multi-GPU version 46

5. Evaluation of CARE 49
5.1. Datasets 50
5.2. Training of Random Forests 51
5.3. Variation of program settings 52
5.4. Evaluation on simulated HiSeq datasets 55
5.5. Evaluation on simulated MiSeq datasets 57
5.6. Evaluation on real-world datasets 58
5.6.1. K-mer evaluation 58
5.6.2. De-novo assembly evaluation 59

6. Performance of CARE 61
6.1. Construction phase 62
6.2. Correction phase 63
6.3. Merge phase 65
6.3.1. Overall performance 66
6.4. Performance with random forests 67
6.5. Read placement 68
6.6. Performance comparison to other tools 72
6.7. Multi-GPU Performance 73
6.7.1. Dataset R2 74
6.7.2. Dataset A4 78
6.8. Proof-of-concept: Sequence parsing on the GPU 81

7. Conclusion 85

II. Read extension 87

8. CAREx: Context-aware read extension 89
8.1. Related Work 90
8.2. Algorithm 90
8.2.1. Construction phase 91
8.2.2. Extension phase 92
8.3. Implementation 96

9. Evaluation of CAREx 99
9.1. Datasets 100
9.2. Program options 101

9.3. Evaluation on simulated datasets 101
9.4. Evaluation on real-world datasets 105
9.4.1. Edit statistics 106
9.4.2. De-novo assembly 107
9.5. Impact of sequencing errors 109
9.6. Performance 110

10. CAREx Conclusion **113**

III. Conclusion **115**

11. Future work **117**

12. Conclusion **119**

Bibliography **121**

List of Figures **129**

List of Tables **133**

A. Appendix **135**
A.1. CARE: Results for simulated data 135
A.2. CARE: K-mer evaluation results for real-world data 137
A.3. CARE: Assembly results for real-world data 139

1 Introduction

1.1 Motivation

Bioinformatics is an important research topic with impact to our daily life as it is found, amongst others, in drug design, personalized medical treatment, vaccine development, and genome studies. At the start of this millennium, the publication of the first fully assembled human genome with over 3 billion base pairs has been a scientific milestone and has provided invaluable insights, for example the layout of chromosomes, the size of coding region, and the number of genes [@31, 1, 11].

Many applications such as genome assemblers operate on DNA snippets which are also called reads. These reads are produced by sequencing machines. Whereas in 2001 the sequencing of a human genome cost an astonishingly $\$100,000,000$ as shown in Figure 1.1, technological advances in the field, next generation sequencing (NGS) and third-generation sequencing (TGS), have led to both a significant increase in sequencing throughput as well as a significant reduction in cost. Nowadays, the cost of a human genome is only a few hundred dollars and it can be sequenced within a day.

One important part of sequencing read processing pipelines is the preprocessing of data to improve the results of down-stream applications. This may involve several quality control options such as trimming, filtering, and/or data modifications like error correction and pseudo-read construction. Trimming removes adapter nucleotides from the sequences that were added during the sequencing process. Sequences may be filtered by a quality value given by the sequencing machine to remove unreliable reads that are of poor quality. Error-correction aims at the removal of sequencing errors that originate from the sequencing procedure. Lastly, paired-end sequencing allows for the construction of a longer sequence, a pseudo-long read, from a pair of sequences. This process is also called read extension.

With the ever increasing amount of data produced by sequencing machines [1], Bioinformatics can be considered Big-Data [77, 68]. Thus, scientists do not only need

[1] The world-wide sequencing capacity in 2015 was around 30 Peta-basepairs. More than 1 Exa-basepairs are estimated for 2025 [77]

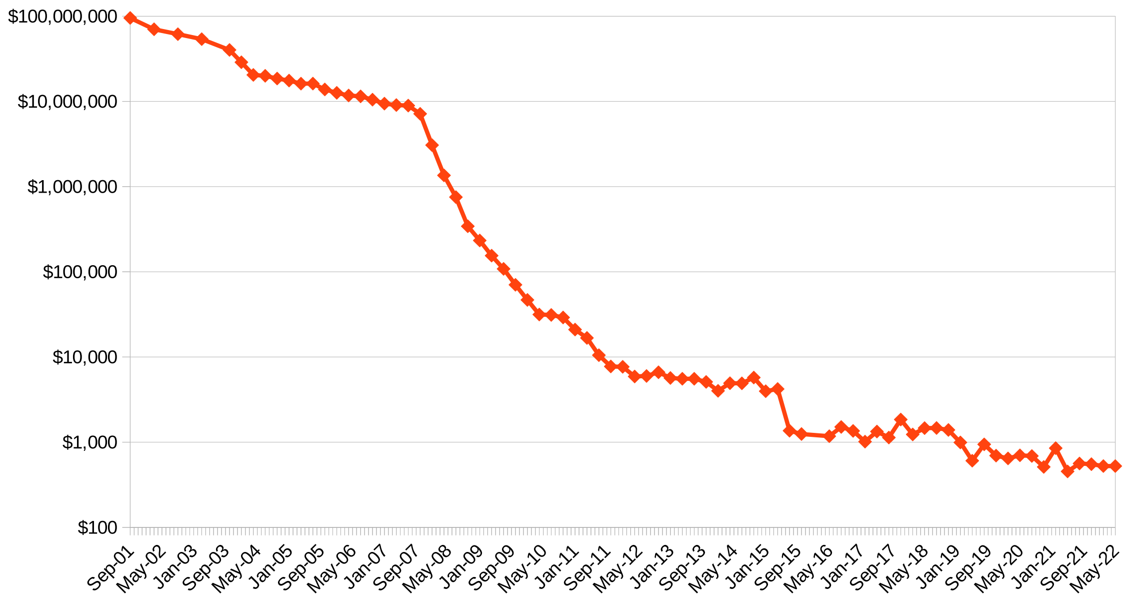


Fig. 1.1.: Sequencing costs per human genome as reported by the National Human Genome Research Institute [@81]

accurate algorithms, but also efficient implementations of those which can cope with the large data volume and produce results in a reasonable amount of time. Traditionally, algorithms have been implemented in programs which execute on the general purpose central processing unit (CPU) of a computer. The presence of multiple compute cores within a CPU allows to execute multiple computations in parallel to reduce processing times. However, depending on the type of workload, this *parallelization* may also be achieved by the use of specialized hardware accelerators like Field-Programmable Gate Arrays (FPGA) and Graphics Processing Units (GPUs). Due to their highly parallel architectures they can thus provide a greater processing power compared to CPUs. Yet, it is not straightforward to harness this performance. Single-threaded algorithms and data structures designed for CPUs are often unsuited for a parallel computing environment. In addition, hardware accelerators typically come with less memory than ordinary CPU systems. Thus, traditional approaches may need to be reformulated, and new parallelized building blocks need to be developed which specifically allow for high-performance execution on those accelerators.

1.2 Thesis structure

In this thesis, we present two novel algorithms for error correction and read extension of NGS data, CARE and CAREx, which have been implemented targeting modern single-node work-stations with GPU support. Chapter 2 highlights relevant biological concepts for this thesis, and introduces sequence similarity and CUDA GPU programming. An overview of related bioinformatic software is given in Chapter 3. Chapters 4 - 7 present our novel GPU-accelerated error corrector CARE. Our approach to read extension with CAREx is explained in Chapters 8 - 10. At last, Chapters 11 and 12 give a final conclusion and thoughts on future works.

1.3 Publications

CARE, our GPU-accelerated MSA-based approach to DNA error correction was published in [34]. It was superseded by CARE 2.0 [33] which showed how a machine learning approach with Random Forests reduces false-positive corrections of CARE. CARE is publicly available under `https://github.com/fkallen/CARE`. We have integrated the CARE algorithm into the quality control software RabbitQCPlus 2.0 [82].

2 Background

2.1 Bioinformatics

DNA is a molecule made of two complementary strands of nucleotides which form a double helix. Each strand has a defined direction from the so called $5'$ end to the so called $3'$ end. The $5'$ end of one strand is complementary to the $3'$ end of the other strand. From a computer science perspective, a DNA sequence (of a single strand) is commonly described as a string over a four-letter alphabet $\Sigma = \{A, C, G, T\}$, representing the four bases of the DNA's nucleotides **A**denine, **C**ytosine, **G**uanine, and **T**hymine. The alphabet may be extended by a fifth letter N, indicating that the base is not unambiguously determined, or simply unknown. The reverse complement of a DNA string, i.e. the complementary strand in its correct reading direction, is obtained by reversing it and replacing each base with its complement base. The complement bases of A, C, G, and T are T, G, C, and A, respectively. In bioinformatics, one often processes short substrings of length k of a DNA sequence, so called k-mers. Given a k-mer and its reverse complement, the canonical representation of a k-mer is the lexicographically smaller sequence.

Given that DNA sequences can be considered strings, they can be stored as plain ASCII files. A common plain-text format for sequences is the FASTA/FASTQ format. These files contain DNA records, so called *reads*. A read consists of a header, the DNA sequence, and, in the case of FASTQ, quality scores. In this thesis, the sequence of a read may also be referred to as read for simplicity. Quality scores have the same length as the sequence. They represent confidence values of sequencing machines, indicating the probability that the base at the corresponding position was called incorrectly, known as Phred Quality Scores [15]. Given the error-probability P, the Phred quality score Q is computed as $Q = -10 \cdot \log_{10}(P)$, which can subsequently be converted into an ASCII character.

To obtain reads from a given biological sample, it needs to be sequenced. Over the last decades, three major approaches to DNA sequencing have been developed, often referred to as Sanger Sequencing / First Generation Sequencing, Next Generation Sequencing / Second Generation Sequencing, and Third Generation Sequencing. First Generation Sequencing describes the methods based on the work of Sanger [66].

They can produce reads of roughly 1000 base pairs (bp) with an error rate of 0.001% . Next Generation Sequencing (NGS) technologies are high-throughput methods which are able to process millions of DNA fragments simultaneously, producing reads of length around 150 bp with an error rate of 0.1%. Last, Third Generation Sequencing refers to sequencing platforms that are able to produce significantly longer reads than previous generations, reaching lengths on the order of $\mathcal{O}(10^4)$. A drawback of those reads, however, is an increased error rate compared to NGS reads. Our work focuses on the processing of NGS reads produced by Illumina machines. There can be two kinds of reads, single-end reads and paired-end reads. With single-end sequencing, a single read is obtained per DNA fragment which can originate from any of the two strands. In paired-end sequencing, two reads are produced per DNA fragment, each from a different strand and from a different end of the fragment. The outer distance between the two reads is called *insert size*. The concept of paired-end sequencing is shown in Figure 2.1.

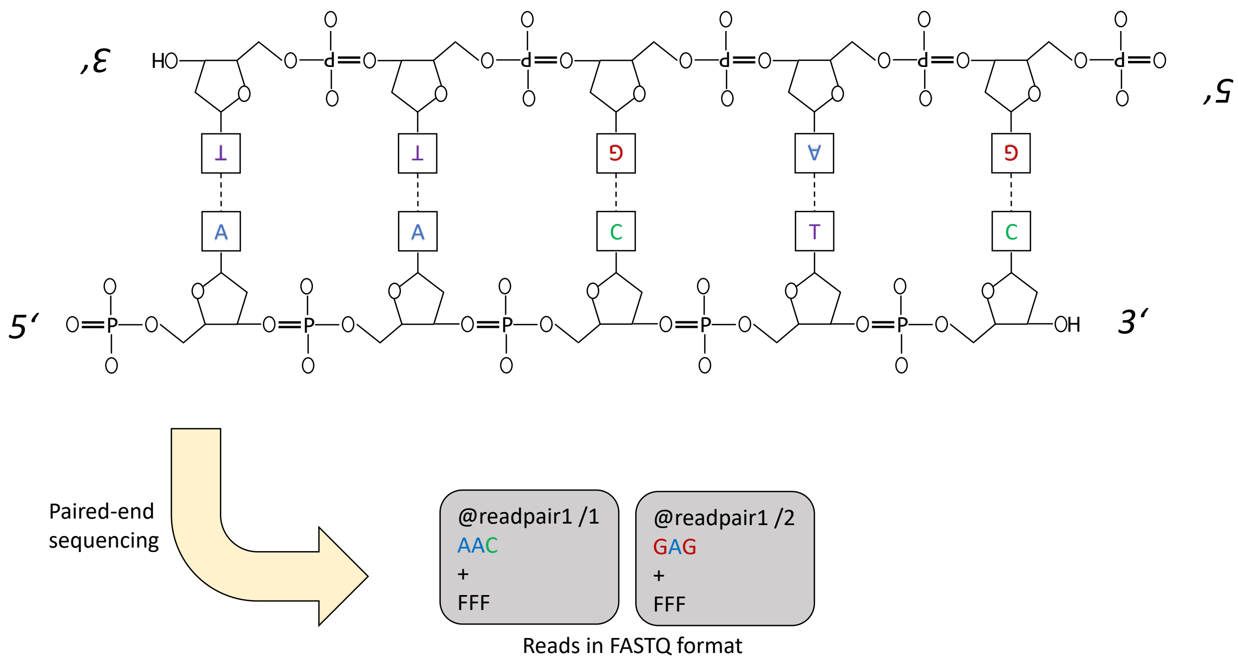


Fig. 2.1.: Paired-end sequencing of a DNA fragment. Typically, not the full fragment length is sequenced per strand.

Another important aspect of sequencing is the depth of coverage. The dataset coverage can be computed as $\frac{readlength \cdot numreads}{genomesize}$. It describes the average number of reads that cover the same position in the genome assuming uniform sequencing of the whole genome. A typical coverage is 30. A dataset with 30-fold coverage of a human genome, which has a length of around $3 \cdot 10^9$ basepairs, could comprise of, for

example, $900,000,000$ reads of length 100. Note that datasets may have non-uniform coverage distributions, i.e. different genome regions may have different sequencing depths.

2.2 Sequence similarity and sequence alignment

A common problem in bioinformatic applications is to decide whether two or more sequences are similar to each other or not. More generally speaking one seeks to assign a similarity score to a set of sequences. This section introduces scoring functions relevant to this thesis, namely hamming distance and sequence alignment. The computation of a hamming distance is simple and fast, but may fall short to alignment algorithms in the bioinformatic context.

2.2.1 Hamming Distance

Let $S1$ and $S2$ be two strings of equal length n over the same alphabet. Then the hamming distance $H(S1, S2)$ is defined as the number of positions with mismatching characters between $S1$ and $S2$. The hamming distance can be computed in $\mathcal{O}(n)$.

$$H(S1, S2) = \sum_{i=0}^{n-1} (S1[i] \neq S2[i])$$

For example, the hamming distance between `CCGATA` and `CGTTAA` is 4. There is a mismatch at position $1, 2, 3,$ and 4.

2.2.2 Pair-wise sequence alignment

A shortcoming of the hamming distance is the assumption that the i-th character of one sequence must correspond to the i-th character of the other sequence. In general, this may not be the case. In bioinformatics both natural biological mutations and artificial errors caused by the sequencing process can introduce insertions and / or deletions (so called indels) into a sequence, i.e characters could be added or removed. Thus, a different algorithm is required to compute accurate similarity scores in the presence of indels to allow for a comparison between $S1[i]$ and $S2[j]$ where $i \neq j$. One such metric which includes indels is the Levenshtein-distance, also known as edit-distance. The edit-distance of a pair of sequences $S1$ and $S2$ is

the minimum number of modifications which need to be applied to $S1$ to obtain $S2$. A modification is either a character substitution, a character insertion, or a character deletion. For example, the sequence `CCGATA` can be transformed into `CGTTAA` using a minimum of three modifications: one deletion, one substitution, and one insertion.

$$\texttt{CCGATA} \rightarrow \texttt{CGATA} \rightarrow \texttt{CGTTA} \rightarrow \texttt{CGTTAA}$$

The sequence alignment algorithm with linear gap penalty is a generalization of the edit-distance algorithm which allows for arbitrary penalty costs of matches, substitutions, and indels. In contrast, the edit distance uses a fixed cost of 1 for substitutions and indels, and a cost of 0 for matching characters. When a cost function is used the best alignment is the alignment with the smallest total costs. Equivalently, a sequence alignment can be computed using a scoring function instead of a cost function. In this case, the optimal solution has the greatest total score. Sequence alignment problems are described by a set of recurrence relations which can be solved using dynamic programming.

Global alignment

Let $S1$ and $S2$ be two strings of length $l1 = |S1|$ and $l2 = |S2|$, respectively, over the same alphabet Σ. Let $\delta : (\Sigma \cup \{-\}, \Sigma \cup \{-\}) \rightarrow \mathbb{Z}$ be a scoring function. A global sequence alignment $A(S1, S2, \delta)$ is a $2 \times k$ matrix over $\Sigma \cup \{-\}$ with two rows and k columns where $\max(l1, l2) \leq k \leq l1 + l2$. The alignment score is then calculated as sum of function δ applied to the two characters of each column.

An example alignment of sequences $S1 =$ `CCGATACAGCTG` and $S2 =$ `CGTTAACTG` with a score of 9 is given by the following table. It uses a score of 2 for matches, -1 for mismatches, and -2 for gaps.

S1	C	C	G	A	T	A	C	A	G	C	T	G
S2	-	C	G	T	T	A	-	A	-	C	T	G
$\delta(S1[i], S2[i])$	-2	2	2	-1	2	2	-2	2	-2	2	2	2

In general, the optimal global sequence alignment score $o_{i,j}$ of prefixes $S1[1:i]$ and $S2[1:j]$ is defined by the following recurrence relation.

$$o_{i,j} = \max \begin{cases} o_{i-1,j} + \delta(S1[i], -) \\ o_{i,j-1} + \delta(-, S2[j]) \\ o_{i-1,j-1} + \delta(S1[i], S2[j]) \end{cases}$$

It is initialized with values $o_{0,0} = 0$, $o_{i,0} = \sum_{x=1}^{i} \delta(S1[x], -)$ for $1 \leq i \leq l1$, and $o_{0,j} = \sum_{y=1}^{j} \delta(-, S2[y])$ for $1 \leq j \leq l2$. The optimal alignment score for the alignment of the full strings is given by $o_{l1,l2}$. This recurrence can be solved in $\mathcal{O}(l1 \cdot l2)$ with dynamic programming using the Needleman-Wunsch algorithm.

Semi-global alignment

Sometimes one wishes to align a sequence to the interior of a significantly longer sequence, for example in the context of read mapping. This could result in many gaps added to the begin and to the end of the shorter sequence because of the length difference. The computation of a global alignment will penalize each of those gaps which may easily dominate an otherwise good alignment with many matches. Similar issues may arise when considering an alignment of a suffix of one sequence to a prefix of another sequence. Since gaps will be added after the end of the suffix sequence and in front of the begin of the prefix sequence, the alignment score may not accurately represent the region of interest. The semi-global alignment solves those issues by assigning a penalty of 0 to all gaps that occur before the first sequence character, and after the last sequence character, respectively. This is achieved by modifications to the global alignment recurrence. First, $o_{i,0} = 0$, $o_{0,j} = 0 \; \forall i, j$. This change allows free gaps at the begin to reach the region of interest without costs. Then the optimal semi-global alignment score between $S1$ and $S2$ is computed as $\max(\max_{0 \leq i \leq l1}(o_{i,l2}), \max_{0 \leq j \leq l2}(o_{l1,j}))$. By computing the maximum of those values, penalizing gaps at the end of sequences are ignored. For example, using the same scoring function δ as in the previous example, an optimal semi-global alignment of `TAGTT` and TTT has a score of 4. The corresponding global alignment score would be -1.

S1	T	A	G	T	T	-
S2	-	-	-	T	T	T
$\delta(S1[i], S2[i])$	0	0	0	2	2	0

2.2.3 Multiple-sequence alignment

The concept of pair-wise sequence alignment can be generalized to more than two sequences. Assume for brevity that all sequences are of same length n. A multiple-sequence alignment (MSA) of m sequences is a $m \times k$ matrix with m rows and k columns where $n \leq k \leq m \cdot n$. Its computation can be expressed as a recurrence relation analogous to the previous pair-wise alignments, where the scoring function δ takes m inputs, each input taken from a different sequence. Following the same dynamic programming approach, an optimal MSA can then be computed in $\mathcal{O}((2^m - 1) \cdot (n^m))$ using a generalized Needleman-Wunsch algorithm. The exponential runtime makes it infeasible to compute exact MSAs of a larger collection of sequences. Additionally, the problem of finding an optimal MSA has been proven NP-hard [8]. This implies that it is unknown whether a polynomial-time algorithm for MSA computation exists. However, a polynomial-time approximation algorithm is known. The STAR approximation algorithm [19] constructs MSAs with approximation ratio of 2 in $\mathcal{O}(m^2 \cdot n^2)$. Briefly, the algorithm works as follows.

1. Compute an optimal pair-wise alignment for each pair of sequences $\mathcal{O}(m^2 \cdot n^2)$
2. Select an anchor sequence A which aligns the best to all other sequences $\mathcal{O}(m^2)$
3. Use the $m - 1$ pair-wise alignments of A to progressively construct the MSA with m rows. $\mathcal{O}(m \cdot n)$

A simple application of an MSA would be the computation of a consensus string. The consensus string consists of the most frequent character of each column. In general, MSAs can be used to detect patterns common to multiple sequences which cannot be determined by simple pair-wise alignments. This can be utilized, for example, for phylogenetic analysis and the detection of genetic mutations such as single nucleotide polymorphisms (SNP). Figure 2.2 shows an example MSA.

S1	-	C	A	G	G	T	C
S2	-	G	A	G	G	-	A
S3	-	G	A	C	T	C	A
S4	-	T	A	C	G	-	A
S5	A	G	A	T	G	-	-
Consensus	-	G	A	G	G	-	A

A	0.2	0	1.0	0	0	0	0.6
C	0	0.2	0	0.4	0	0.2	0.2
G	0	0.6	0	0.4	0.8	0	0
T	0	0.2	0	0.2	0.2	0.2	0
-	0.8	0	0	0	0	0.6	0.2

Fig. 2.2.: An MSA of five sequences with its consensus string and its probability profile.

2.3 GPU computing

Graphics Processing Units (GPUs) are co-processors which have been traditionally used to accelerate computer graphic operations. General Purpose Computation on Graphics Processing Unit (GPGPU) allows us to utilize the available hardware resources for custom operations with a high degree of parallelism. The main differences between CPUs and GPUs lie in the number of cores and the type of accessible memory. CPUs typically come with a few tens of cores. Current high-end server CPUs may have 128 cores like an AMD EPYC 9754 processor. In contrast, both consumer-grade and server-grade high-end GPUs, for example NVIDIA RTX 4090, NVIDIA H100, or AMD Instinct MI250X, consist of more than ten thousand cores. In terms of memory, GPUs have access to faster memory such as high bandwidth memory (HBM) with transfer rates of over 3 TB/s on an NVIDIA H100 GPU, but the amount of memory on a single graphics card is limited to around 80GB, although multiple cards may be used to increase the total available memory. On the other hand, CPU memory like DDR5 memory provides much smaller transfer rates, but

server-scale boards can fit more than 1TB of memory. For example, a memory bandwidth of 460 GB/s can be achieved by a single AMD EPYC 9754 CPU which supports up to 6 TB of memory.

GPUs can be programmed with different frameworks. OpenCL provides a general API for heterogeneous programming. OpenCL code can be executed on different kinds of hardware as long as vendor-specific drivers exist, which is the case for, amongst others, Intel CPUs, AMD CPUs, AMD GPUs, and NVIDIA GPUs. NVIDIA CUDA is a programming language that targets exclusively NVIDIA GPUs. AMD's counterpart to CUDA is its ROCm platform that supports both AMD CPUs and GPUs. In addition, it provides the high-level API HIP which can be translated to OpenCL targeting AMD hardware, or translated to CUDA targeting NVIDIA hardware.

In this thesis, we used CUDA to parallelize our software. However, the general concepts of our parallelization schemes can be applied in any of the APIs mentioned above. In the remainder of this section, we will give a brief introduction into the CUDA programming model and how CUDA programs are executed on hardware. For more detailed information, we refer to the official CUDA programming guide [@56].

CUDA Programming model

CUDA C++ is an extension to the C++ programming language that allows to write C++ code which can be executed on NVIDIA GPUs. It is compiled by specific compilers such as NVCC. CUDA adds execution space identifiers `__host__`, `__device__`, `__host__ __device__`, and `__global__`, that can be used to annotate functions. Functions without annotation or with annotation `__host__` can only be called from CPU code. Functions with `__device__` can only be called from GPU code. Specifying both indicates that the function can be called on both CPU and GPU. We call code that executes on the CPU host code. Device code is executed on the GPU.

The entry point to GPU code is given by functions annotated with `__global__`, so called kernels. Kernels are functions that are called via CUDA API on the host, but are executed on the device with a user-specified number of threads. The GPU driver is responsible for executing the function on the device. The number of specified threads is allowed to be greater than the number of threads that can physically be executed in parallel on the hardware. Kernel execution is asynchronous with respect to the host. To wait for the completion of a kernel, host code must make a synchronizing CUDA API call, for example `cudaDeviceSynchronize`, which blocks execution of the host thread until the GPU finished its processing.

The device-accessible GPU memory can be logically separated into three memory spaces, global memory, shared memory, and local memory. Local memory is only accessible by a single kernel thread. Shared memory is accessible by groups of kernel threads. All kernel threads can access global memory. Typically, GPU and CPU use distinct memory address spaces which means ordinary CPU memory is not accessible by device code, and vice-versa. This requires the programmer to manually allocate GPU-accessible memory and explicitly transfer data to it for kernel inputs and retrieve data from it to obtain the kernel outputs. `cudaMalloc` and `cudaMemcpy` are functions that can be used from the host to allocate GPU memory in the global memory space, and to transfer data to and from this memory, respectively. Both functions have asynchronous versions that may return before the operation in question is completed, in the same manner as kernel launches. In addition, both operations may execute in parallel to kernels which are executing on the GPU. This allows for complete overlap between GPU kernels, data transfers, and host-side work. Shared memory and local memory cannot be modified from the host.

GPU threads in a kernel can execute independently and are hierarchically organized. At the lowest level there are the individual threads. 32 consecutive threads are called a warp. A thread block consists of up to 32 warps which is equal to 1024 threads per block. At the top level, there is the grid which contains all thread blocks. The Hopper GPU architecture added support for an intermediate level between thread blocks and the grid, the so called thread block cluster which groups consecutive thread blocks. CUDA provides built-in variables which allow to compute a unique thread identifier for all threads in the kernel in all levels of the thread hierarchy. When a kernel is called from the host, it is mandatory to specify the total number of thread blocks in the grid, as well as the total number of threads per thread block.

Within each level of the thread hierarchy, threads can communicate, collaborate, and synchronize, with efficiency decreasing towards the top of the hierarchy. Threads within a warp execute in SIMT fashion (single instruction, multiple threads). They are able to efficiently exchange data amongst each other via so called warp-shuffle operations. The function `__syncwarp` acts as a barrier for all threads within a warp, i.e. it synchronizes their execution. The function `__syncthreads` synchronizes all threads within a thread block. Threads belonging to different warps within the same thread block can exchange data via the shared memory which is slower than warp shuffles, but faster than accessing device-wide global memory. Shared memory is shared between all threads within a thread block. Apart from communication between threads, it can be used as fast scratch-pad memory or a manually managed cache for frequently used data. All thread blocks have at least 48 KB of shared memory available. The upper limit is dependent on the GPU architecture. The

largest amount of shared memory per thread block is provided by the newest Hopper architecture (227 KB). Thread blocks have access to the shared memory of the other thread blocks within the same thread block cluster. Threads in different thread block clusters can only communicate and be synchronized via global memory, which is less efficient than shared memory-based cooperation within a thread block or thread block cluster.

CUDA can make use of multiple GPUs to further increase processing capabilities and available device memory. Data locality plays an important role in multi-GPU computations. For trivially parallelizable algorithms where each GPU can hold its required data the performance scales linearly with the number of GPUs. In general, however, data dependencies between GPUs lead to performance penalties. While each GPU is able to access data resident on a different GPU, those remote device memory accesses, enabled by hardware interconnects, are several times slower than local accesses. This places a burden on developers to find the best distribution of data amongst the GPUs, and may require the use of communication collectives like broadcast and all-to-all to efficiently move data between GPUs. Large-scale datasets in bioinformatics may not fit into the memory of a single GPU and either must be stored in ordinary host memory and be transferred on demand, or be distributed amongst multiple GPUs. Although memory requirements can be reduced by employing techniques like data compression, dealing with large datasets remains challenging.

To give an example of a parallel algorithm, we take a brief look at a parallel reduction. It is commonly used to compute the sum of all input values. Figure 2.3 shows the concepts of a block-wide parallel summation assuming four threads per warp. Each warp loads a chunk of input data into registers. Each thread can access data of other threads within the warp via shuffle operations to incrementally compute a warp-wide partial sum. The partial sums of each warp are stored in shared memory. A synchronization barrier ensures that each warp stored its sum before a single warp computes the final block-wide sum. To extend this approach to a grid-wide reduction (without considering thread block clusters), each thread block needs to store its partial sum to global memory. After a grid-wide synchronization barrier, a single thread block can load the intermediate results and computes the final result.

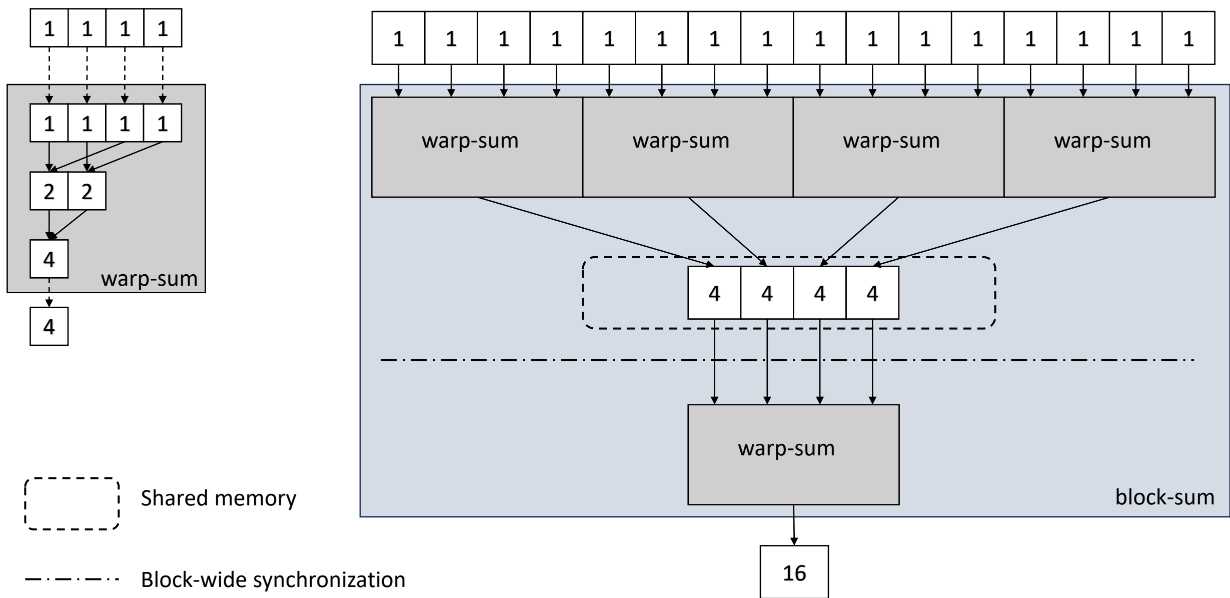


Fig. 2.3.: A block-wide parallel reduction

CUDA hardware implementation

While the hardware implementation exists separately from the programming model, achieving best performance in GPU programming requires knowledge of the underlying hardware, similar to ordinary CPU programming. From a hardware perspective, kernels are executed on NVIDIA GPUs by Streaming Multiprocessors (SMs). At its heart an SM consists of data processing units, a register file comprising of 65536 32-bit registers, and a unified data cache which serves as shared memory for thread blocks as well as an L1 cache. Multiple SMs share a portion of the GPU's L2 cache. The software-side thread blocks are assigned to SMs for execution. The scheduling order of thread blocks to SMs is unspecified. Multiple thread blocks can be executed on an SM in parallel. When a thread block finishes execution, the next thread block is assigned to the SM. The software-side global and local memory spaces reside in off-chip DRAM. DRAM accesses are cached in the L2 cache. The shared memory space corresponds the shared memory provided by an SM.

The number of thread blocks which can be executed in parallel on one SM is limited by the available hardware resources. To give an example, assume that the number of threads per block is 256 and each thread requires 128 32-bit registers for execution. Then at most $65536/(256 * 128) = 2$ thread blocks can be executed in parallel on the

same SM. In the same manner, another upper limit can be derived from the available shared memory per SM.

Recall that thread blocks are partitioned into warps of size 32. At each GPU clock cycle, warp schedulers select a number of warps of the assigned thread blocks to execute their next instruction. Since the whole execution state of a warp is stored in the large register file, execution of different warps can be interleaved via zero-overhead context switches. This is crucial for high throughput as it allows to schedule instructions from a different warp while another warp may be stalled on a data dependency or execution dependency. The maximum number of warps that can issue an instruction per clock cycle is hardware-dependent.

3 Related Work

With continuous sequencing efforts in recent decades, error correction is not a new research topic but has been studied for many years. One important use-case for error correction is genome assembly [60] where a genome is constructed from sequencing reads. Genome assemblers often perform error correction with an integrated algorithm [75, 4, 50, 84] but there also exist many stand-alone programs for the error correction of DNA sequences which may also be used in different bioinformatic pipelines that are susceptible to sequencing errors, for example, genotyping and variant calling.

Here, we will give a brief overview of error correctors and their employed data-structures. A summary is given in Table 3.1. A more detailed survey of tools and strategies can be found, for example, in [38].

Name	Data structure	Target error type
	NGS	
SGA [75]	FM-index	substitutions
Musket [46]	BF+HT	substitutions
RACER [25]	HT	substitutions
Lighter [76]	BF	substitutions
BFC [39]	BF+HT	substitutions
BLESS [20]	BF+HT	substitutions
RECKONER [13]	BF+HT	substitutions
HiTEC [26]	SA	substitutions
Bcool [42]	DBG	substitutions
DecGPU [48]	BF	substitutions
Blue [17]	HT	substitutions + indels
Coral [65]	HT	substitutions + indels
ECHO [35]	HT	substitutions + indels
Fiona [69]	SA	substitutions + indels
Karect [2]	SA	substitutions + indels
	TGS	
FMLRC [80] (hybrid)	FM-index	substitutions + indels
LoRDEC [64] (hybrid)	DBG	substitutions + indels

Tab. 3.1.: Selection of error correction tools for NGS data and TGS data with their type of corrected errors and the key utilized data-structure: Bloom filter (BF), Hash table (HT), Suffix Array (SA), FM-index, and De Bruijn graph (DBG).

A frequently used approach in error correction, for example by Musket and BFC, utilizes hash tables and / or bloom filters to derive a "ground truth" from the dataset which is subsequently used as a reference to modify the dataset in concordance with the ground truth. Hash tables can also be used for similarity searches, for example in Coral or Echo, to find similar reads which could be subsequently used in a consensus-based error correction. Other approaches may use text matching techniques to index the whole dataset to allow for efficient searches of common patterns. HiTEC and Karect employ a suffix tree whereas SGA constructs an FM-index. Bcool follows the approach of genome assemblers and uses a De Bruijn graph to derive corrected sequences. One of the earliest GPU-accelerated error correctors is CUDA-EC [72] which uses a bloom filter to reduce memory consumption. DecGPU uses a combination of CUDA and MPI for parallel distributed error correction of large datasets to overcome the memory limitations of a single GPU, which still poses a problem today. Machine learning is used in many different areas of bioinformatics, for example in protein folding [30], medical image classification [41], and DNA language modeling [29]. In the context of DNA error correction, LERNA [71] uses a transformer-based language model for DNA to aid the selection of program parameters of error corrector. To the best of our knowledge, we are one of the first to use machine learning within a DNA error correction algorithm.

From Table 3.1 it is also apparent that there are different types of sequencing errors. Error-correction software may specialize in the processing of different types of errors. NGS data contains orders-of-magnitude fewer insertions and deletions (indels) than substitutions. Thus, many tools only focus on substitutions. Nonetheless, there are algorithms which consider the presence of indels. The ability to correct indels becomes especially important for TGS reads. The approaches for TGS error correction typically involve self-correction [10] or a hybrid correction. With hybrid correction, employed for example in FMLRC and LoRDEC, higher-quality NGS reads are mapped to the TGS reads to perform a consensus-based correction. Self-correction only uses the present TGS reads for correction. This is often achieved by computing pair-wise alignments between the reads and constructing multiple-sequence alignments.

Many applications need to quickly identify similar sequences, for example to construct a multiple-sequence alignment. The brute-force approach of comparing all pairs of sequences in a dataset typically is computationally infeasible due to the size of datasets which can contain millions of sequences. More efficient similarity searches based on short substrings of fixed length can be implemented with look-up tables. In the most simple case, all such substrings are generated for all sequences and inserted into a table that can be queried to find sequences which contain specific substrings. These match locations can then be inspected further. However, storing

all substrings may require vast amounts of memory. Memory requirements can be reduced by using sub-sampling techniques. A group of substrings could be represented by a minimizer [63] which is the lexicographically smallest substring in the group. Then, only those minimizers are stored in the lookup table. Minhashing is a specific locality-sensitive hashing sub-sampling technique that was originally introduced by search engines to detect near duplicate web pages [9]. In minhashing, a hash value is computed for each substring within a group. Then, the h smallest hash values, called a signature, represent the group, and allow the approximation of the Jaccard index between two groups of substrings. In recent years, minhashing has gained popularity for processing NGS data with examples including genome assembly [6], metagenomics [58, 54], and read mapping [61]. Our work utilizes a variant of minhashing. We believe to be the first to adapt this concept in the context of error correction.

When performing genome analysis aided by sequence data, in general only structures smaller than the read length can be resolved correctly. A drawback of NGS reads compared to Sanger reads or TGS reads is their relatively short length (~ 150bp (NGS), ~ 1000bp (Sanger), $> 10,000$bp (TGS)) which complicates or prohibits the identification of larger structures within a genome. This can lead to issues in the detection of structural variants [52] or in the treatment of repeat regions during genome assembly. Conceptually, genome assemblers already perform read extension since the resulting contigs are super-strings of the reads from which longer reads could be extracted. In fact, many tools for read extension employ techniques used in genome assembly, read overlapping and De Bruijn graphs, to produce longer reads. For example, one could identify similar reads which subsequently can be overlapped and merged to create a longer read. ELOPER [73] and GapFiller [55] realize this approach by using hash tables. Konnector2 [79] constructs a De Bruijn graph of the whole dataset and performs a graph traversal to extract longer sequences.

Many of the previously mentioned tools come with limitations in terms of result quality and / or processing speed. In terms of quality, a problem which can observed for error correction is high numbers of newly introduced errors (false positive corrections). For read extension, some tools may only be able to process a small fraction of the input. Performance problems include bad scaling with input data size, insufficient parallel efficiency, or even lack of parallelization. Also, the vast majority of tools are purely CPU-based. While this is not a problem per se, these tools do not leverage the opportunity of faster processing on additional hardware. For relevant datasets such as Human datasets with 30-fold coverage, this leads to runtimes that can range from hours to days. For example, the error correctors Musket and BFC take 3 hours and 1.5 hours, respectively, on a CPU with 64 threads. Read extender

Konnector2 takes over a day to complete. In contrast, our proposed error corrector CARE and read extender CAREx complete in 30 minutes and 3.3 hours, respectively, on a single A100 GPU. CARE is further able to utilize multi-GPU systems to reduce the runtime to 10 minutes.

Besides error correction and read extension, many other areas of NGS processing benefit from the fast processing on GPUs for compute-intensive applications. NVIDIA Clara Parabricks [@57] is a software suite that provides GPU-accelerated implementations of commonly used CPU-based programs for NGS read analysis such as read mapping or variant calling. Google's DeepVariant [62] is a deep-learning based variant caller with GPU support. CUDA ClustalW [24] and GPU-ClustalW [45] provide a parallel implementation of the well known ClustalW algorithm for progressive multiple-sequence alignment construction [78]. Short-read alignment and similarity searches are accelerated in Cushaw [47], SOAP3 [44], and CUDASW++ [49]. Fast meta-genomic classification is provided by MetaCache-GPU [36].

Part I

Sequencing read error correction

4 CARE: Context-Aware Read Error correction

Modern sequencing technologies can produce high-coverage datasets consisting of many millions or even billions of short sequencing reads. Produced reads, however, are not perfect but are affected by noise which manifests in the form of sequencing errors. Such errors can affect down-stream analysis in a negative way. Error correction software is often employed to remove many of these sequencing errors making it an important building block in sequence processing pipelines including genome assembly [21] and SNP calling [16].

CARE, context-aware read error correction, is an MSA-based error corrector for NGS HiSeq Illumina reads. Given a collection of n reads, a multiple-sequence alignment is constructed per read, the so called anchor read. This MSA contains the anchor read which should be corrected, as well as other reads which are assumed to be similar to that read. Then, a corrected anchor read can be extracted from the MSA using its consensus information.

In a naive approach, finding all sets of similar reads in a collection of n reads requires the computation of $\mathcal{O}(n^2)$ pair-wise sequence alignments, discarding those alignments with a low similarity score. However, with datasets consisting of millions of reads, this naive approach is infeasible. CARE relies on hash tables to find potentially similar reads instead of computing all pair-wise alignments. This is done in a two-phase approach with a hashing scheme similar to minhashing. First, hash values of all reads are stored in a database, i.e. hash tables. Then, a set of potentially similar reads can be found by querying this database for the hash values of an anchor read. In a follow-up operation, pair-wise alignments are computed between the anchor read and potentially similar reads to discard dissimilar reads. While this approach may not find all similar reads in the collection of reads, it is preferred over the naive approach because the number of similar reads is significantly less than n, i.e. the total number of required alignment computations is reduced drastically.

4.1 Publications

Parts of the work presented in Chapters 4, 5, and 6 have been published in the following peer-reviewed papers. Some text fragments, examples and results are directly taken from the papers. This thesis gives additional implementation details and includes a multi-GPU implementation. Chapter 5 extends the results presented in the papers. Chapter 6 investigates the performance of different parallelization approaches and compares the runtimes of CARE to its competitors.

CARE: context-aware sequencing read error correction.
Felix Kallenborn, Andreas Hildebrandt, Bertil Schmidt.
Bioinformatics, Volume 37, Issue 7, March 2021, Pages 889–895.
`https://doi.org/10.1093/bioinformatics/btaa738`

CARE 2.0: reducing false-positive sequencing error corrections using machine learning.
Felix Kallenborn, Julian Cascitti, Bertil Schmidt.
BMC Bioinformatics volume 23, Article number: 227 (2022).
`https://doi.org/10.1186/s12859-022-04754-3`

RabbitQCPlus 2.0: More Efficient and Versatile Quality Control for Sequencing Data.
Lifeng Yan, Zekun Yin, Hao Zhang, et al.
Methods, Volume 216, August 2023, Pages 39-50.
`https://doi.org/10.1186/s12859-022-04754-3`

Abstract – Next-generation sequencing pipelines often perform error correction as a preprocessing step to obtain cleaned input data. State-of-the-art error correction programs are able to reliably detect and correct the majority of sequencing errors. However, they also introduce new errors by making false-positive corrections. These correction mistakes can have negative impact on downstream analysis, such as k-mer statistics, de-novo assembly, and variant calling. This motivates the need for more precise error correction tools. We present CARE 2.0, a context-aware read error correction tool based on multiple-sequence alignment targeting Illumina datasets. In addition to a number of newly introduced optimizations its most significant change is the replacement of CARE 1.0's hand-crafted correction conditions with a novel classifier based on random decision forests trained on Illumina data. This results in

up to two orders-of-magnitude fewer false-positive corrections compared to other state-of-the-art error correction software. At the same time, CARE 2.0 is able to achieve high numbers of true-positive corrections comparable to its competitors. On a simulated full human dataset with 914M reads CARE 2.0 generates only 1.2M false positives (FPs) (and 801.4M true positives (TPs)) at a highly competitive runtime while the best corrections achieved by other state-of-the-art tools contain at least 3.9M FPs and at most 814.5M TPs. Better de-novo assembly and improved k-mer analysis show the applicability of CARE 2.0 to real-world data. False-positive corrections can negatively influence down-stream analysis. The precision of CARE 2.0 greatly reduces the number of those corrections compared to other state-of-the-art programs including BFC, Karect, Musket, Bcool, SGA, and Lighter. Thus, higher-quality datasets are produced which improve k-mer analysis and de-novo assembly in real-world datasets which demonstrates the applicability of machine learning techniques in the context of sequencing read error correction.

4.2 Related Work

Current state-of-the-art error correctors are often classified by their underlying algorithmic approach into k-mer-based and MSA-based methods.

4.2.1 K-mer-based methods

A k-mer is a substring of length k of a genomic sequence. In a k-mer-based approach the k-mer spectrum of a collection of sequencing reads is inspected to identify k-mers which are error-free with high confidence, so-called *solid* k-mers. k-mers which are not solid are called *weak*. Often, k-mers are distinguished as solid or weak based on their frequency in the dataset given a supplied frequency threshold. k-mers which reach the threshold are considered solid. k-mer-based error correction algorithms typically try to replace weak k-mers by similar solid k-mers. Figure 4.1 gives an example of k-mer-based error correction.

While this approach is simple and fast, it usually suffers from a great number of false-positive (FP) corrections because low frequency correct, but weak, k-mers may be changed into erroneous, but solid k-mers which appear more often. Due to its simplicity, this approach is used in many error correction tools such as SGA-EC [75], Musket [46], RACER [25], Lighter [76], Blue [17], BFC [39], BLESS 2 [20], and RECKONER [13].

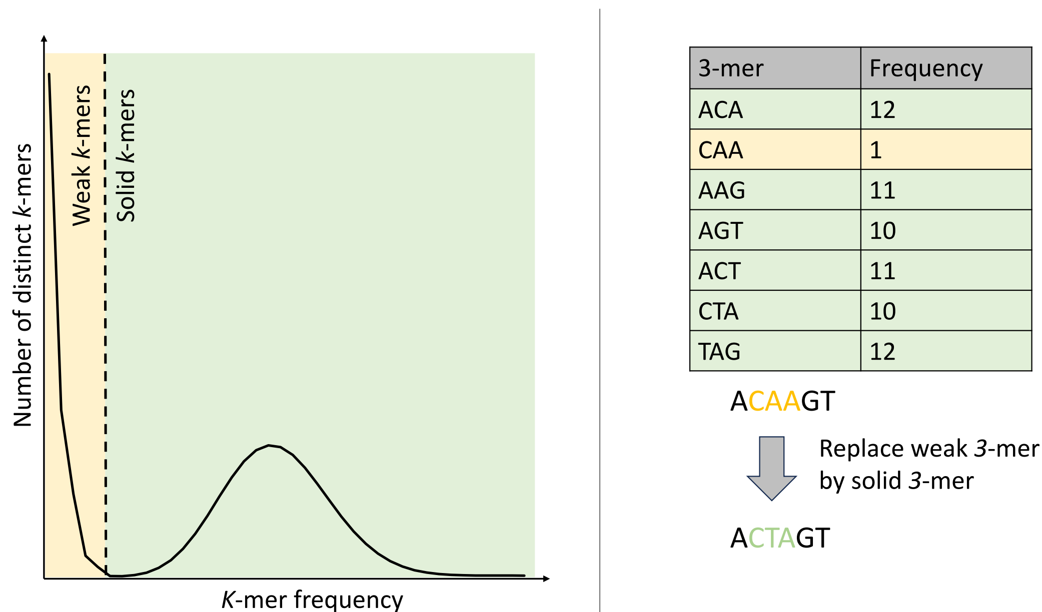


Fig. 4.1.: Left: k-mer frequency histogram. A frequency threshold separates weak k-mers (yellow) from solid k-mers (green). Right: k-mer based modification of a sequence.

4.2.2 MSA-based methods

MSA-based algorithms identify groups of similar sequences and arrange them in a multiple-sequence alignment (MSA). In contrast to changing individual k-mers in isolation, MSA-based error correction utilizes the additional information contained in the MSA, such as per-column coverage, and sequence contents of positions surrounding a potentially erroneous position. This typically allows for higher error correction precision. However, a major drawback of the MSA-based approach is its high computational complexity to construct MSAs. The first MSA-based error correctors specifically designed for Illumina data were Coral [65] and ECHO [35]. More recent examples of alignment-based error correctors are Fiona [69], Karect [2], Bcool [42], BrownieCorrector [22].

4.3 Algorithm

The CARE algorithm takes an input file with sequences in FASTA or FASTQ format and produces an output file consisting of corrected reads. There are three processing steps: construction phase, correction phase, and output phase. In the construction phase, the input sequences are loaded into memory, and hash tables are constructed

from the sequences. The correction phase is the core of the algorithm. Here, corrected reads are computed. Finally, in the output phase an output file is constructed from the corrected reads. The three steps will now be explained in detail.

4.3.1 Construction phase

The goal of the construction phase is to load the input reads from file into memory, and to provide populated hash tables of those reads for similarity search.

CARE uses a variant of minhashing to hash the n reads. The hashing process is configured by two parameters, the number of hash functions and hash tables h, and the k-mer size k. Let $H = f_1, \ldots, f_h$ be a set of h hash functions. For each read r_i, a read signature S is computed from the canonical k-mers of read r_i. This signature consists of h hash values. $S[m]$ is computed as the smallest observed hash value with hash function f_m. Note that more than one value of S could be computed from the same k-mer of r_i because hash functions are applied independently. Finally, the key-value pair $(S[m], i)$ is inserted into the m-th hash table. After construction of the hash tables, they store information about which reads share at least one common hash value. This is a good indicator that those reads share a common k-mer. However, because of hash collisions this may not always be the case.

4.3.2 Correction phase

During the correction phase each read of the input data is processed to compute a corrected read. The correction of a read depends on the redundant information in the dataset. It is quantified by the dataset coverage c which is the average sampling depth of each position in the genome, i.e. each position of the genome is covered by c reads on average. For high values of c, a single error at a specific position can then be identified by a simple majority vote. For example, if 1 out of 30 samples contains a different nucleotide, that one nucleotide could be replaced by that of the remaining 29 samples. Such a majority vote requires that all participating reads originate from the same location. To find those reads, the previously constructed hash tables are utilized, followed by several filters to remove bad reads. In the remainder of this section, it is explained how input read r_i is corrected. r_i is also referred to as anchor read. Figure 4.2 shows the workflow of the algorithm.

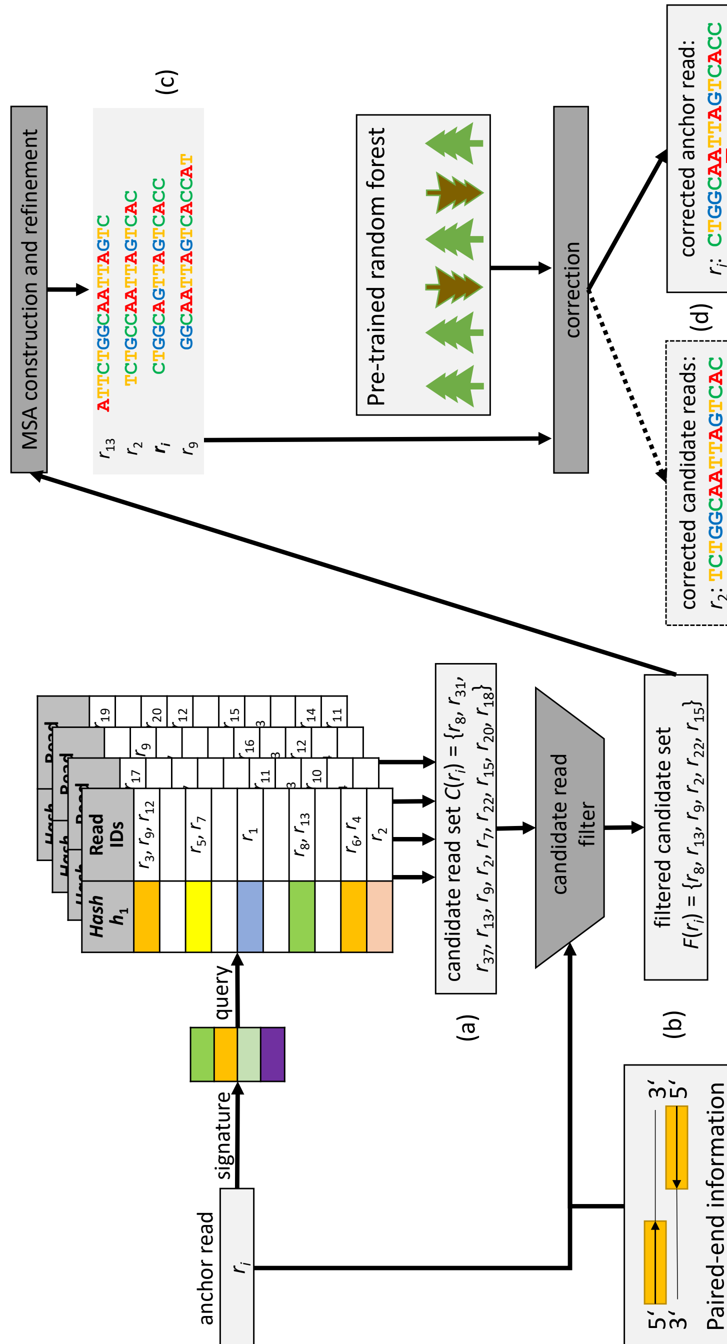


Fig. 4.2.: Workflow of CARE: (a) The signature of an anchor read (r_i) is determined by minhashing and used to query the precomputed hash tables. The retrieved reads form the candidate read set $C(r_i)$. (b) All reads in $C(r_i)$ are aligned to r_i. Reads with a relatively low semi-global pair-wise alignment quality are removed, resulting in the filtered set of candidate reads ($F(r_i)$). (c) The initial MSA is constructed around the center r_i using $F(r_i)$. The MSA is refined by removing candidate reads with a significantly different pattern from the anchor (i.e. r_{15}, r_{22}, r_7 in the example). (d) The anchor read (the seventh nucleotide in r_i in the example) and optionally some of the candidates are corrected (the fifth nucleotide in r_2 in the example), using a provided random forest trained for correction.

Computation of an initial candidate set

At first, the read hash signature S of size h is computed a second time. The hash value $S[m]$ is queried in hash table m. Each query yields a set of ids of reads which share the same hash value at position m of their respective read signature. Let C_i be the union of all h query results. The reads (or read ids, to be more precise) in C_i are called candidate reads.

Subsequently, each candidate read is aligned to the anchor read to identify candidate reads which are similar to r_i and overlap r_i by at least 30% of positions. In addition, the alignment indicates their relative orientation to r_i, i.e. whether the candidate read and the anchor read are located on the same DNA strand or on complementary strands. The latter information is found by computing two alignments per candidate and using the better one, where in one alignment the reverse complement sequence of the candidate is used instead of the forward sequence. In general, those similarity calculations would require the computation of semi-global alignments to allow for free shifts to arrange the candidate, and to account for gaps in a sequence, i.e. insertions or deletions (indels). However, CARE targets reads produced by the Illumina platform where the dominant types of sequencing errors are mismatches, not indels [5, 67]. Thus, only an alignment with free shifts is considered that also only accounts for mismatches. This is implemented as an efficient shifted hamming distance calculation using bit-wise operations.

The alignment computation returns four values per candidate:

- the alignment orientation (forward or reverse complement)
- the number of overlapping positions
- the number of mismatches in overlapping positions
- an optimal shift value

The shift value indicates by how many positions the candidate read needs to be shifted to the right relative to the anchor read to obtain the other three values. Negative shift values correspond to left-shifts. If the shift value s is positive, the first letter of the candidate will overlap with the s-th letter of the anchor. If the shift value is negative, the s-th letter of the candidate will overlap with the first letter of the anchor.

Computation of an filtered candidate set

After alignments have been computed, candidates are filtered by their alignment results. A perfect filter would only keep candidates which originate from the same genomic location as the anchor read. Candidates whose alignment contains many mismatches are discarded, as well as candidates which do not sufficiently overlap the anchor read. CARE provides two different filter implementations. The used implementation depends on whether the input dataset should be treated as single-end or paired-end. In the single-end implementation, candidates are assigned to four bins depending on the relative number of mismatches per overlapping position of the alignment, i.e. $\frac{\text{mismatches}}{\text{overlapsize}}$. The first bin contains candidates with up to 6% mismatches, the second bin those with up to 12%, and the third bin allows a mismatch rate of up to 18%. All candidates fit the last bin. Note that a candidate can be assigned to multiple bins. Next, the bin with the smallest index is computed whose number of assigned candidates reaches a threshold of $0.6 \cdot$ estimated dataset coverage. Candidates which are not assigned to the selected bin are removed by the filter.

When using the paired-end mode, both reads of a read pair are processed simultaneously by the correction algorithm. After finding the two candidate sets and computing the respective pair-wise alignments with the two anchors, the filter can utilize the pair information. The two candidate sets are inspected to find candidates of the same read pair, where one of the reads is a candidate of one anchor, and the mate of that candidate, i.e. the other read of the pair, is a candidate of the anchor's mate. Such pairs of candidates are kept in the candidate sets unconditionally. CARE assumes that this condition holds true only if the anchor read pair and the candidate read pair both originate from the same genomic region. Candidates which do not fulfill this condition are filtered by the ratio between number of mismatches and size of overlap. Instead of a binned approach, a simple threshold is applied. Candidates with a ratio greater than t_{paired} (default: 0.06) are removed.

As an example, assume candidates of both an anchor read a_0 and its mate a_1 should be filtered. Let the candidate sets of a_0 and a_1 be $C(a_0) = \{r_0, r_5, r_{11}\}$ and $C(a_1) = \{r_4, r_8, r_{14}\}$, respectively. Two consecutive reads $r_{2 \cdot i}, r_{2 \cdot i+1}, i \in \mathbb{N}$ form a read pair. Then r_4 and r_5 always pass the filter, because they originate from the same read pair. The remaining candidates are kept depending on their alignment quality to the corresponding anchor read.

Construction of an MSA

Let F be the set of candidate reads which passed the alignment filter. Those candidates are assumed to be similar to the anchor read and can be used in a majority vote to correct the anchor. To do so, anchor read and filtered candidate reads are arranged in a multiple-sequence alignment M. This is achieved using the previously calculated pair-wise sequence alignments in a manner similar to the STAR algorithms for constructing approximate MSAs, using the anchor read as the center sequence. Since gap-free alignments are used, the construction of an MSA is straightforward. Assume that there are $n = |F|$ candidates, and all sequences involved are of equal length l. The corresponding MSA can be described by a matrix with $n + 1$ rows and at most $l \cdot 2.4$ columns. The upper limit of the number of columns stems from the minimum required overlap of 30% between anchor and a candidate.

CARE does not store the full matrix of M since the ordering of rows is irrelevant. Thus, only the counts of each type of nucleotide (A,C,G,T) as well as their weights are stored per column, which saves memory. The nucleotide weight at position x of a candidate is determined by the alignment quality, and the read quality scores. The weight is a floating point number in the range $[0, 1]$. The contribution of alignment quality is the same for each x and is computed from the number of mismatches per overlapping positions. Longer overlaps with fewer mismatches correspond to greater weights. The quality score weight for position x is given by 1 minus the error probability value encoded by the quality score string at position x. The final nucleotide weight is the product of alignment weight and quality weight. When no quality scores are used, the quality weight is set to 1.0. For anchors, the alignment weight is set to 1.0. Given the counts and weights, the following MSA attributes can be derived for each column: coverage, consensus, and support.

The *coverage* of a column is the sum of its nucleotide counts. The *consensus* of a column is the nucleotide with the greatest weight. The consensus string of M is the concatenation of every column consensus. The error correction process will try to replace nucleotides in the anchor by the consensus nucleotide of its corresponding column in M. The column *support* is the relative weight of its consensus, i.e. consensus weight divided by the sum of nucleotide weights.

Additionally, for columns that are occupied by the anchor, counts and weights of the anchor nucleotides are stored separately. Let x be the nucleotide of the anchor at a specific position. Then, the so called *original coverage* is given by the count of x

in the corresponding MSA column. Analogously, the *original weight* is given by the weight of nucleotide x in that column.

After M has been constructed, its column contents are inspected to find patterns which make precise error correction more difficult. This approach is called refinement. MSA refinement is an iterative process which is repeated at most five times. It aims to remove candidates from F and M which may originate from an inexact repeat region similar to the region of the anchor. We try to identify those candidates by searching for columns in M which are occupied by the anchor read and consist of any non-consensus nucleotide x which contributes at least 30% of the estimated dataset coverage to the column coverage. Let a be the anchor nucleotide and x as defined above. If $a = x$, candidates without nucleotide x in the selected column are marked for removal. Else, those candidates are marked which do have nucleotide x in the selected column. Next, alignments of marked candidates are checked. If there is no marked candidate with an alignment quality weight of at least 0.9, marked candidates are removed from F and M which completes one refinement iteration. If, on the other hand, at least one alignment quality weight of a marked candidate reaches 0.9, no candidate is removed. This results in an early exit of MSA refinement.

Standard read correction

At this stage of the algorithm to correct anchor read r_i, set F contains candidates which share a hash value with r_i, have a good alignment to the anchor, and can thus be used to produce a reliable correction of the anchor. First, the constructed refined multiple-sequence alignment M is classified as either high-quality or low-quality depending on the minimum coverage, average support and minimum support of the columns in M which are occupied by the anchor. Then, the consensus string of M is used to provide a corrected anchor sequence. Note that it is possible to obtain a corrected sequence which is equal to r_i. In the case of high-quality M, each position of the anchor is unconditionally replaced by the corresponding column consensus nucleotide of M. For low-quality MSAs, however, a more selective replacement strategy is employed which does only modify positions of high confidence. Positions with both a column support greater than 0.9 and a nucleotide frequency of at most 2 for the anchor nucleotide are considered as high confidence positions.

In addition to the anchor, a high-quality multiple-sequence alignment can be used to correct its candidates, as well. The candidates in F align well to the anchor and are assumed to originate from the same genomic region. Since the quality

of M is determined by the contents of anchor columns, its consensus can also be applied to candidates if they align to the anchor with a shift value of small magnitude. Specifically, a candidate correction is produced from the MSA consensus for each candidate which is fully contained in the MSA column range $[l - x, r + x]$, where l and r are the left-most column and right-most column occupied by the anchor, respectively, and $x = 15$ is the default value for candidate correction. The consensus is used unconditionally for each occupied position of the candidate. The range is chosen close around the anchor because its columns have the greatest coverage. Coverage decreases towards the left end and the right end of M. A small column coverage can make the column consensus less reliable, especially since MSA refinement is not applied to columns not covered by the anchor. Note that, in total, there can be multiple corrections of the same read as candidate of different anchors. The set of all produced candidate corrections of anchor r_i is later used to either confirm or reject the produced anchor correction of r_i.

Random Forest-based read correction

CARE implements a second, alternative correction mode that utilizes Random Forests for corrections. It can be enabled via a program argument. The anchor correction with high-quality MSAs remains unchanged. For low-quality MSAs, a pre-trained Random Forest is used. The anchor sequence is compared to the consensus string to identify mismatching positions. For each of those positions, a set of position-dependent features is extracted from the MSA which are subsequently classified via the Random Forest. Given the features, the Random Forest decides whether the nucleotide at that specific anchor position should be replaced by the consensus of the corresponding column in the M.

The same approach can be applied to candidate corrections. Instead of using the consensus unconditionally for a candidate correction as in the standard correction mode, a pre-trained Random Forest is used to find the positions that should be replaced by the consensus nucleotide.

Random Forest-based anchor correction and candidate correction require two separate forests with separate features. This is because of the different MSA structures. For anchor corrections, the Random Forest is applied to low-quality MSAs, whereas in the case of candidate corrections the Random Forest operates on high-quality MSAs. Detailed information about the random forests is given in Section 4.4.7.

4.3.3 Output phase

After the construction phase is completed, for each input read there exists one anchor correction and zero or more candidate corrections. Now, a single corrected read needs to be constructed per input read from the cached anchor corrections and candidate corrections.

First, the list of all corrections is sorted by read id. The set of all corrections of r_i is then merged to produce a final correction which is written to the output file. The merge process depends on both the quality of the MSA which generated the anchor correction, and the number of candidate corrections. If it was a high-quality MSA, the anchor correction is the final correction. For low-quality MSAs, we consider three cases which depend on the available candidate corrections. Recall that candidate corrections are obtained from high-quality MSAs. If there are at least two candidate correction, they are used to verify the (maybe erroneous) anchor correction. If all candidate corrections are equal to the anchor correction, it is the final correction, else it is rejected and read r_i remains uncorrected, i.e. unchanged. In the case of at most one candidate correction, the anchor correction is always selected as the final result.

4.4 Implementation

There exist two implementations of the CARE algorithm for Linux systems, a purely CPU-based version written in C++, and a GPU-based version written in CUDA/C++. Both versions utilize host-side multi-threading to improve the performance. In addition, the GPU version can offload work to CUDA-enabled GPUs which allows for highly-parallel execution. In the GPU version, both sequence data and hash tables can either reside in host memory or device memory.

4.4.1 Data structures

CARE organizes its input data and hash tables in two data structures. Access to sequence data is managed by a *read-storage*. A *minhasher* is used to query the hash tables. Both data structures are represented by abstract base classes. Derived classes exist for the CPU version and the GPU version, respectively. A third data structure, which is used in both versions of CARE, provides temporary storage to cache corrected sequences.

Read-storage

The read-storage uses scatter / gather semantics to access sequence data and quality scores by read id. Given a number i from 0 to $N-1$ where N is the number of reads in the dataset, the stored data of the i-th read can either be gathered into a target buffer, or modified by scattering data from a buffer. The concrete implementations use arrays to store the data. For the GPU read-storage implementation, distributed arrays are used where the data of a specific read can either reside in host-memory or in the device-memory of one GPU. All implementations support batch operations.

An input read consists of one or two strings, depending on the input file format. If the input file is in FASTA format, only the DNA sequence is given per read. If the input file is in FASTQ format, the DNA sequence is accompanied by a quality score string of same length. These strings, as well as their lengths, need to be stored in memory for efficient processing.

In general, DNA sequences are represented as strings over a four-letter alphabet {A,C,G,T}. Occasionally, the letter N can be observed as well in a small fraction of sequences. N stands for an ambiguous, undetermined nucleotide, and can be either A,C,G, or T. In CARE, each occurrence of N is deterministic replaced by either A,C,G or T. Then, to reduce memory consumption by up to 75%, those DNA sequences are converted to a two-bit representation, where $A = 00_2$, $C = 01_2$, $G = 10_2$, and $T = 11_2$. Let l_{max} be the greatest observed sequence length. It is determined in the construction phase. Then each 2-bit encoded sequence occupies $\lceil l_{max}/16 \rceil \cdot 4$ bytes in the read-storage (16 2-bit letters per 4 byte integer). Shorter sequences are padded with zeros to this length but typically all sequences have similar lengths, for example $100-102$ bp.

Read lengths are stored in a compact format. For each read, its length is represented by an offset which needs to be added to the smallest read length observed in the dataset. Then, the offsets are stored in a contiguous bit array without padding, using the smallest number of bits per offset required to represent the range of offsets. For example, assume 4 reads with read length 101, 100, 105, and 102, respectively, and their corresponding offsets 1, 0, 5, and 2, respectively. Then, $\lceil \log_2(5) \rceil = 3$ bits will be used per offset and the contents of the bit array will be $001'000'101'010$. In the special case that each read is of the same length no offsets are stored.

Quality scores in theory can represent every 8-bit ASCII character. However, depending on the sequencing platform, quality scores usually use only a subset of ASCII characters. For example, the current Illumina platforms use 94 out of the 256 possible characters (33 "!" to 126 "$\sim$"). Yet, this data range does not allow for

an easy, efficient lossless bit-wise encoding. Thus, quality scores are commonly stored without modification. This is the default case for CARE, which means that quality scores can consume up to 80% of the memory required for the input dataset. However, a lossy compression of quality scores is possible by the use of quality score binning [@27]. Given a (short) list of predefined bins and assignment of ASCII values to bins, individual quality scores of a read are replaced by the score of the its assigned bin. Then, the binned quality scores can be efficiently represented by a bit-wise encoding similar to that of sequences. In CARE, the number of quality bins can be 2,4, or 256, where a value of 256 corresponds to the default, uncompressed storage format. This means that compressed quality scores occupy one bit, or two bits, per position, respectively. Compared to the default storage format, this allows to reduce the memory usage for quality scores by a factor of up to 8. Note that the use of quality scores can be disabled. In that case, no quality scores are stored and all reads are assumed to have the same quality.

Minhasher

The minhasher interface provides two methods. First, the query result size can be determined for a given sequence. Second, a given sequence can be queried against the hash tables. There exist three minhasher implementations, one for the CPU version and two for the GPU version. In both GPU implementations, hash values are computed on the GPU. The implementations can be distinguished by the placement of hash tables, which can either reside exclusively in host memory, or exclusively in device memory. All implementations support batch operations.

The hash tables in CARE require a significant amount of memory. Every minhasher implementation attempts to reduce the memory footprint of hash tables. The maximum number of values per key is set to $2.5 \cdot c$ where c is the dataset coverage. All key-value pairs are removed whose key appears more than $2.5 \cdot c$ times. This is motivated by the possible existence of reads originating from inexact repeat regions in the genome. K-mers of those regions are assumed to occur more frequently. Similar reads in such a region cannot be reliably corrected because there exist multiple correction possibilities. Thus, hash values which are assumed to correspond to those regions are removed. To further reduce their memory footprint, hash tables are stored in a compact format. All values are stored in a contiguous array. A compact open-addressing single-value hash table is then used to map the keys to their corresponding values in the value array.

To be able to create this compact layout, all key-value pairs of a table need to be computed beforehand. Subsequently, they are filtered and arranged in the compact format. This approach temporarily requires enough memory to store all key-value pairs of a table, as well as its compact representation. For CPU hash tables, the initial key-value pairs are first sorted by key. Then the number of values per key is determined. If it exceeds $2.5 \cdot c$, the key and all its values are removed. The remaining values are then stored into a contiguous array as described previously, and the single-value hash table of keys is constructed. In the case of GPU hash tables, key-value pairs are inserted into a device-side multi-value hash table with a bucket size of $(2.5 \cdot c) + 1$. The number of unique keys and the corresponding value counts are determined for all buckets of size less than $(2.5 \cdot c) + 1$. Those keys and values are then arranged into the required compact format. Figure 4.3 presents the workflow for a single hash function with a CPU hash table.

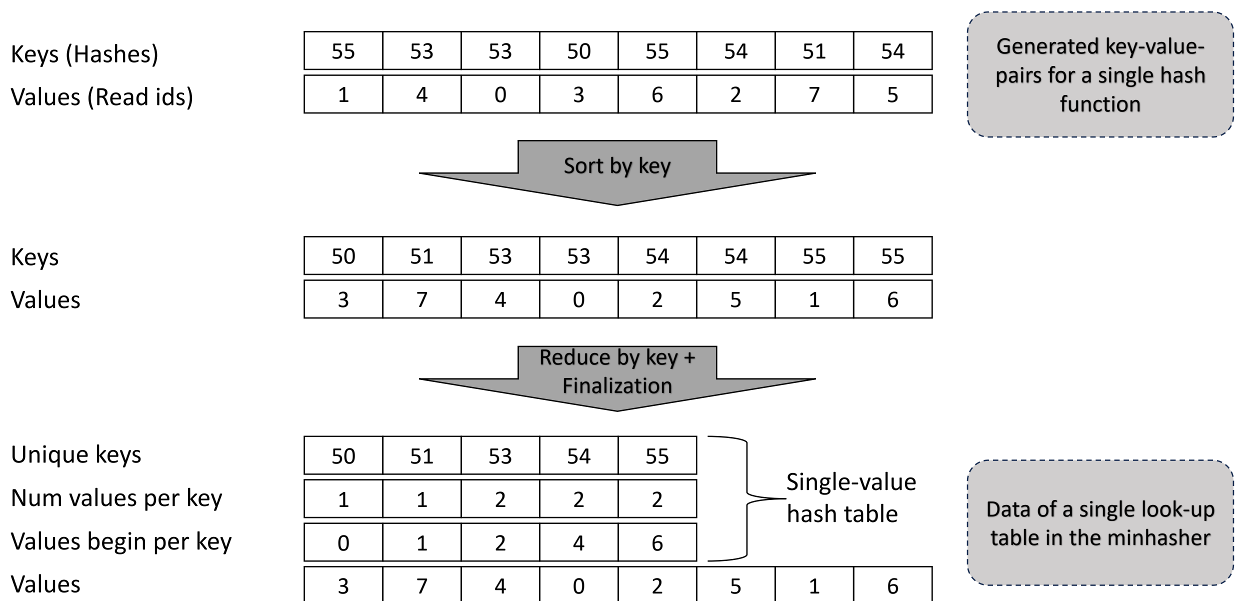


Fig. 4.3.: Constructing a look-up table from the generated key-value pairs of a single hash function.

Temporary storage for corrections

Recall that per read there may be one anchor correction and potentially multiple candidate corrections which need to be stored in the correction phase for access in the output phase. Corrected sequences are stored in a compressed, binary format.

When there are up to 16 modifications made to a read, only the locations of those modifications and their modified nucleotides are stored instead of the full corrected sequence. The corrected sequence is then restored on demand by applying the modifications to the original sequence.

The temporary storage is provided by a contiguous range of virtual memory allocated with the `mmap` system call. The range is contiguous to allow for simple access. However, it is logically split into two sections. The first memory section is backed by physical memory, i.e. RAM. Its maximum size is specified via program argument. The last section is represented by a memory-mapped file that is grown on demand.

4.4.2 CPU version

During the construction phase, reads need to be stored in memory, and hash tables have to be constructed. The first task is achieved by a three-stage pipeline which uses different sets of threads per stage. The input files are parsed in the first stage by a single thread. Reads are converted into a 2-bit storage format in the second stage by four threads. Threads in this stage are also responsible to compress the quality scores, if required. Finally, the converted reads are inserted into a read-storage by a single thread. The three stages run concurrently. Queues are used to communicate between threads of different stages. To parse the sequence file, we implemented a custom FASTA/FASTQ parser in C++ which is based on the kseq C library[1]. The parser supports plain-text files as well as gzip-compressed files.

The second task can be split into two parts, hash table construction and hash table compaction. During construction, batches of reads are hashed in a parallel for-loop which iterates over reads. Subsequently, produced key-value pairs are inserted into hash tables using another parallel for-loop, where each thread is responsible for the pairs belonging to a specific hash table. The compaction of a hash table is enhanced using parallel algorithms from the Thrust library targeting the OpenMP back-end.

The correction of reads is an embarrassingly parallel process. Each read (in the single-end case), or read pair (in the paired-end case), can be corrected independently. Thus, the correction phase is trivially parallelized over the reads to be corrected using OpenMP.

Multiple threads are involved in the construction of the output file. One thread parses the input file. A number of threads decodes the produced anchor corrections and candidate corrections. Another thread consumes input reads and decoded

[1] https://github.com/attractivechaos/klib/blob/master/kseq.h

corrections, and combines them into a final corrected read. The last thread is responsible to write the final corrected reads to the output file.

4.4.3 Single-GPU version

There are two key aspects to achieve good performance in a GPU-accelerated program. First, data locality is important. If possible, frequently accessed data should be stored directly in device memory to avoid expensive data-transfers from host memory via PCIe bus. Second, the GPU should be fully utilized. Often, this requires latency hiding by overlapping GPU workloads with data-transfers or CPU workloads. To achieve data locality, reads are placed on the GPU. In the case that not all of the reads fit into device memory, the excess reads remain in host memory and need to be accessed via PCIe bus.

The GPU version comes in two flavors which differ in the location of hash tables. Hash tables can either be located in host memory, or device memory. The hash tables remain in host memory by default because of their high memory usage. Still, it is possible to opt-in to use GPU hash tables. This may require a majority of available device memory on consumer GPUs even for medium-sized datasets. Our GPU hash tables are based on the GPU hash tables provided by the Warpcore library [32]. During both construction phase and correction phase, hash values of reads are always computed on the device. In the case of CPU hash tables, hash values are subsequently copied to the host to access the tables. The following explanations assume the default case that hash tables are located on the host, not on the device.

During the construction of the read-storage from the input file, the 2-bit encoding of sequences and the optional encoding of quality scores is performed on the GPU. For the construction of hash tables, reads remain in host memory since the hash table compaction is GPU-accelerated using the same Thrust algorithms as in the CPU version, but using the GPU backend. This may require a large amount of GPU-accessible memory. If the available device memory is not sufficient, managed memory is used as a fallback. After construction, the reads are copied to the device.

The correction phase is run almost exclusively on the GPU, with the exception of host hash table lookups. To overcome performance limitations by slow hash table operations, a multi-threaded producer-consumer pattern is used. Reads are processed in batches.

A producer thread gathers the anchor reads of the current batch on the device and computes their hash values. Hash values are then transferred to the host to query

the hash tables. Query results are transferred back to the device. Subsequently, candidate sequences are fetched from the read-storage. The producer finishes processing the batch by submitting it into a work queue.

Consumers fetch batches from the work queue and launch the GPU kernels required for the remaining correction steps, i.e. alignment computation, alignment filtering, MSA construction, and actual correction. Subsequently, correction results are transferred to the host where they are saved in the temporary storage for corrections by a dedicated background thread. By default, up to two consumer threads are used. A subset of the remaining available threads are used as producers. The actual number of producers can be determined dynamically at runtime by measuring the elapsed time of producer workload and the consumer workload for the first few batches, which are executed by a single thread. It is also possible to manually set the thread configurations.

When GPU hash tables are used instead of CPU hash tables, reads are also processed in batches but the described producer-consumer approach is not required since the CPU-intensive hash table operations are no longer performed. Instead, a few independent host threads are used to parallelize over the reads, similar to the CPU version. Each host thread submits batches of reads to the GPU, and uses double-buffering as explained above. Figure 4.4 visualizes the thread configurations for both types of hash tables. Note that depending on the available device memory, the loading of sequences may be a CPU workload if sequences are stored in host memory.

During the output phase, the corrections are sorted by read id using a device-side radix sort algorithm with the option to fall back to a CPU-based sort in case of device-memory shortage. The remaining work is identical to the CPU version.

4.4.4 Batched minhasher queries

In the GPU version, anchor reads are processed in batches for better device utilization. The process of determining the set of candidate read ids for all anchor reads can be split into two steps, hash table queries and post-processing. The query step produces a list of query results per anchor in device memory. A post-processing step transforms these lists into the sets of candidate read ids in device memory. Figure 4.5 shows the corresponding workflow for a batch of two reads.

At first, the minhash signatures are computed on the GPU. This is achieved using one thread per hash value. Next, the tables are queried. In the case of GPU hash tables,

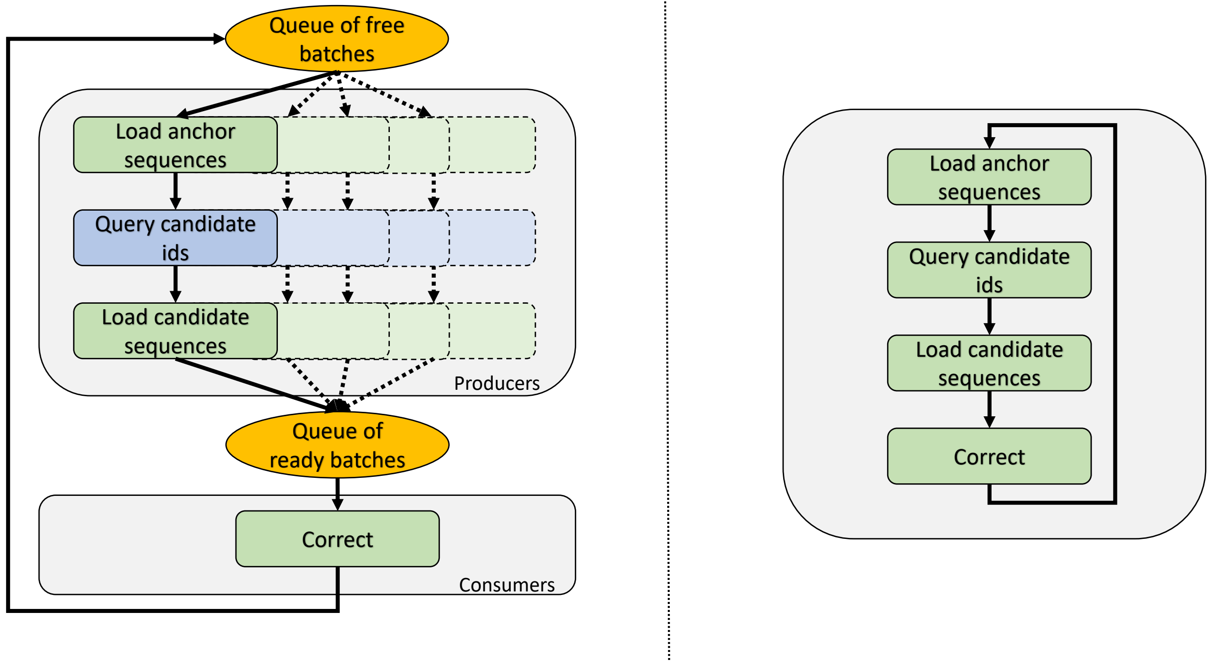


Fig. 4.4.: Producer-consumer-pipeline for CPU hash tables (left) and a simple pipeline for GPU hash tables (right). Blue and green indicate CPU workloads and GPU workloads, respectively.

warpcore uses a sub-warp of 8 threads per hash value. The final candidate read ids are given by the set of distinct values in the result list. To obtain it, each segment is sorted. Subsequently, only the first occurrence of each run of equal elements is kept.

Result lists with more than 2048 elements (large segments) are sorted in global memory using CUB's `DeviceSegmentedSort` function, and are compacted with a custom kernel with one thread block per segment. Result lists with at most 2048 elements (small segments) are processed with a custom kernel that fuses sorting and compaction, using one thread block of size 128 per segment. For those segments, all elements fit into the registers of a thread block which allows to perform an efficient block-wide radix sort with CUB's `BlockRadixSort`. The sorted lists do not need to be materialized in global memory. The compaction step can operate directly on the registers which reduces the number of global memory accesses compared to the large segments. To further improve the performance, we partition the small segments by the number elements and use specialized versions of the kernel for different segment sizes. Specifically, we use 5 different kernels for maximum segment sizes of $128, 256, 512, 1024$, and 2048, where each thread is assigned $1, 2, 4, 8$, or 16 elements

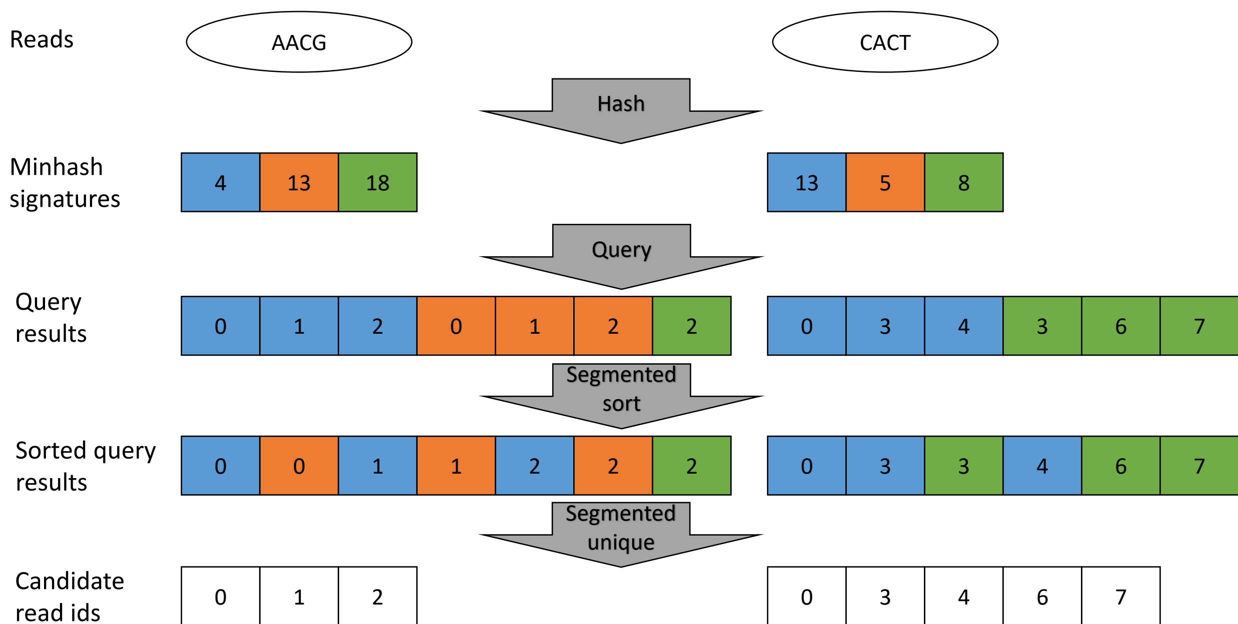


Fig. 4.5.: Finding candidate read ids for a batch of two reads. Three hash functions are used. Different colors indicate results of different hash functions.

of input, respectively. However, segment sizes for both small and large segments can still vary significantly which can lead to load-balancing issues.

4.4.5 Bit-parallel hamming distance

The computation of the hamming distance for different shift values is an important building block of CARE. A single hamming distance computation between two strings from a four-letter alphabet can be performed in bit-parallel fashion.

Recall that sequences in CARE are stored in a 2-bit format with $A = 00_2$, $C = 01_2$, $G = 10_2$, and $T = 11_2$. To compute the hamming distance between 2-bit encoded sequences S_1 and S_2, a bit mask is computed that indicates mismatching characters. This is achieved by first splitting the two corresponding bit strings into two bit strings each, containing all even bits (S_{1e}, S_{2e}) and all odd bits (S_{1o}, S_{2o}), respectively. Next, the bit string $M = (S_{1e} \oplus S_{2e}) \vee (S_{1o} \oplus S_{2o})$ is computed. M is a mismatch mask where the bit at position i is 1 if the characters of S_1 and S_2 at position i are different from each other. Otherwise, the bit is 0. Thus, the hamming distance is given by the number of bits set to 1 in M, also known as the population count. It can be

computed efficiently via intrinsics on capable hardware such as modern CPUs and GPUs. The computation of M for two example sequences is shown in Figure 4.6.

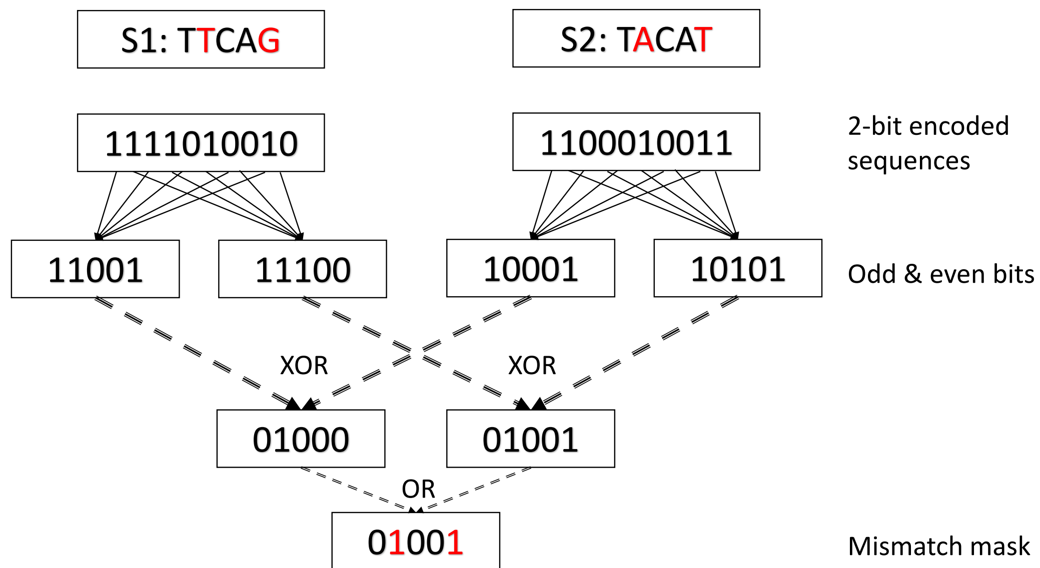


Fig. 4.6.: Bit-parallel hamming distance computation. Mismatching characters are highlighted red.

The initial conversion of the bit-strings is expensive. For a single hamming distance calculation, there is no performance benefit. However, this approach is suitable for CARE since multiple hamming distance calculations are performed per pair of sequences. The converted bit-strings can be reused for multiple shift values.

In the GPU implementation, one pair of sequences is processed per GPU thread. We avoid frequent accesses to slow global memory by loading the sequences once into shared memory. To reduce shared memory bank conflicts, the sequences are stored in block-transposed layout, i.e. the i-th sequence element of thread t is stored in shared memory at address $i \cdot threadsPerBlock + t$ such that consecutive threads in the same block access consecutive memory locations.

4.4.6 MSA construction

Multiple-sequence-alignments are constructed from the anchors, their candidate sequences, and their pair-wise alignment results. Optionally, quality scores can be considered as well. The output consists of multiple arrays describing the column contents of the MSA, such as per-column nucleotide frequency and nucleotide weight.

For the parallel construction of many MSAs on the GPU, we use one thread block with 128 threads per anchor. Since CARE targets short NGS reads, both the four frequency arrays and the four weight arrays fit into shared memory. For example, with sequences of maximum length 128 around 14 kilobyte of shared memory are required. Thus, our MSAs are created in shared memory and are only written back to global memory after the construction is complete.

For the actual population of the arrays, we had initially partitioned the thread block into multiple smaller thread groups, for example size 8, which each processed one candidate sequence and added the counts and weights using atomic operations. The reasoning behind that approach was that 2-bit compressed sequences are stored in few 32-bit integers (8 for sequences of length 128), which could be assigned to individual threads. However, this led to atomic contention for columns of high coverage, and caused non-deterministic results because floating point math (for the summation of weights) is not associative. For example, this caused problems during the MSA refinement process, where, despite the removal of all candidates covering a specific column, the sum of weights for that column could end up with a non-zero value.

These problems were resolved by using the simple approach of processing candidates one after another, using one thread per nucleotide. While this requires a block-wide synchronization after each candidate to avoid race-conditions, the performance improved compared to the atomic approach and floating point issues were eliminated.

4.4.7 Random Forest

Random Forests in CARE perform the binary classification task of deciding whether or not a specific position in a read should be replaced by the consensus. Given the features of a single position extracted from the MSA, each decision tree in the forest produces a confidence value between 0.0 and 1.0. Error correction is performed if the sum of confidence values in the forest exceeds a threshold. As a performance optimization, classification finishes as soon as the threshold is reached, without evaluating every decision tree, if possible.

A feature for Random Forest-based error correction is a floating point number which is derived from the MSA. Table 4.1 lists all features used by CARE. The anchor correction is performed using features 2-14. Candidate correction uses features 1-14.

1	relative overlap between anchor and candidate
2	average support of anchor columns
3	minimum support of anchor columns
4	minimum coverage of anchor columns / c
5	maximum coverage of anchor columns / c
6	count of o / coverage
7	weight of o / full column weight
8	weight of o / count of of o
9	count of x / coverage
10	weight of x / count of x
11	support
12	coverage / c
13	full column weight / c
14	full column weight / coverage

Tab. 4.1.: Features extracted from an MSA. c is the estimated dataset coverage. x is the consensus nucleotide. o is the original nucleotide of the sequence to be corrected (anchor or candidate). Features 6-14 describe only the currently inspected column, whereas features 1-5 are properties that cover multiple columns and are constant for all positions of the same anchor or the same candidate, respectively.

The Random Forests for CARE are trained offline with the scikit-learn python package [59]. For this purpose, the CPU version of CARE comes with the option to extract features from its computed MSAs and to store them to file. The features are then labeled using a ground-truth dataset and are subsequently fed to the training script.

CARE uses a custom implementation and serialization of Random Forests providing a simple integration into the existing code for both the CPU version and the GPU version. Random Forests are loaded from file. Each decision tree is represented as a set of nodes which are stored contiguously in an array. Each node stores the feature number and the corresponding split value. If a node's child is not a leaf, the array index of the child node is stored, as well. If it is a leaf, its confidence value is stored instead. The ordering of nodes within the array is given by the pre-order DFS traversal of the tree. While this choice comes with a sub-optimal performance caused by irregular memory accesses during classification, it is simple to implement and simple to use. Faster, more complex representations exist. For example, [70] suggests a hierarchical partitioning into sub-trees where the nodes of each sub-tree are stored contiguously.

4.4.8 Multi-GPU version

CARE is able to utilize multiple GPUs. This comes with two advantages that reduce the runtime compared to using a single GPU.

First, batches of reads can be processed in parallel on multiple GPUs. This is achieved by splitting a batch into n equal-sized parts where n is the number of GPUs. A single CPU thread is used to submit work to all GPUs in round-robin fashion.

Second, using multiple GPUs extends the available GPU memory, allowing to store more sequence data in device memory instead of host memory. Additionally, more hash tables fit into GPU memory when GPU hash tables are enabled. There are different methods to utilize the extended memory. In the most simple case, all read data and device-side hash tables fit into the memory of a single GPU. Then, the data can be replicated on each GPU, allowing for direct efficient access without additional communication overhead between GPUs. In the case that read data and/or hash tables exceed the available memory of a single GPU, reads and/or tables are distributed amongst multiple GPUs using an even-share distribution. This adds additional inter-GPU communication steps when accessing hash tables or reads. For minhasher access, all input sequences are broadcast to each GPU, and the per-GPU query results are sent back to the respective devices where they are merged into a contiguous array. Gathering reads from the read-storage via an index list requires a multi-split operation to determine the corresponding indices per GPU, followed by an all-to-all communication to distribute those index lists. Figure 4.7 gives an example of multiple GPUs gathering data from a distributed array.

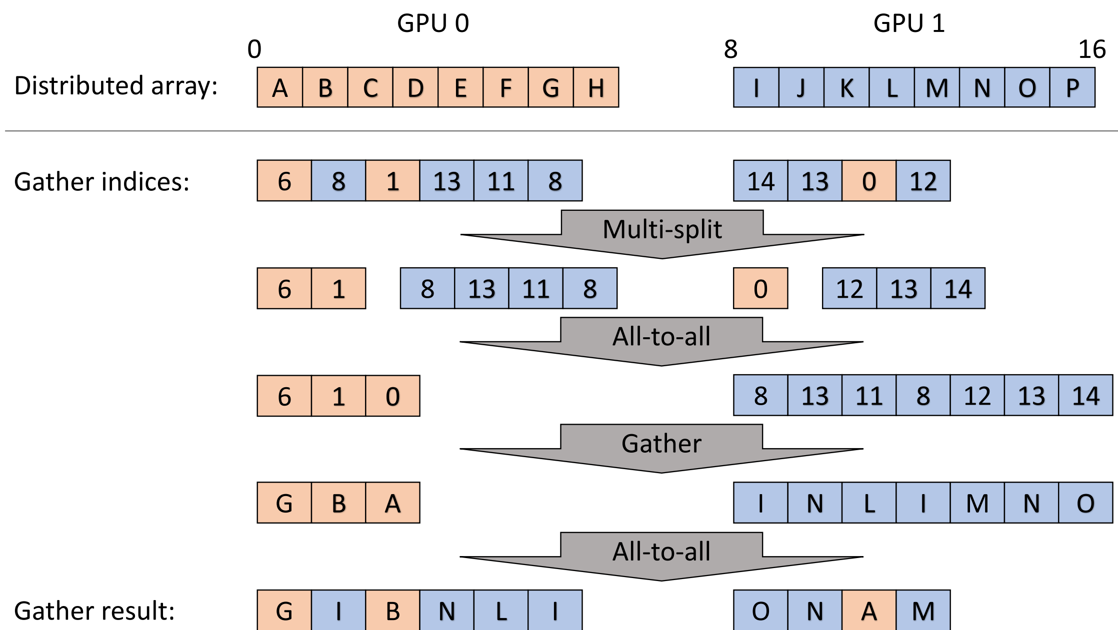


Fig. 4.7.: An array with 16 elements is evenly distributed between two GPUs. The array is collectively accessed by each GPU via index list. Indices per GPU are identified via (non-stable) multi-split and exchanged. Locally gathered data is sent back to the respective GPUs. The second all-to-all operation includes a reordering of gathered data to match the order of input indices.

5 Evaluation of CARE

In this chapter we investigate CARE's ability to detect and correct sequencing errors in FASTQ files. Simulated datasets provide a simple way to analyze the output of different algorithms since the exact numbers and locations of sequencing errors are available, as well as the exact location of reads within a reference genome. In contrast, the erroneous nucleotides and read locations are generally unknown for real-world datasets. Still, evaluation of real-world data is important to investigate whether positive observations in simulated datasets lead to positive observations in real datasets. This may not be the case if simulated datasets do not accurately model error distributions of the target sequencing platforms. CARE was developed targeting reads from the Illumina HiSeq platform.

For simulated datasets with known erroneous positions it is easy to compute per-nucleotide statistics (true positives, false positives, true negatives, and false negatives) by a three-way comparison of the input dataset I, an error-free version of the input dataset E, and the corrected dataset C. This way, it is possible to classify each nucleotide of the correction output as either true positive (TP), false positive (FP), false negative (FN), or true negative (TN). Let I_i, E_i, and C_i be the i-th nucleotide of files I, E, and C, respectively. Then, the following definitions of, TP, FP, FN, and TN, are used.

- TP: $I_i \neq E_i$ and $E_i = C_i$
- FP: $I_i = E_i$ and $E_i \neq C_i$
- FN: $I_i \neq E_i$ and $E_i \neq C_i$
- TN: $I_i = E_i$ and $E_i = C_i$

For this evaluation, the ART read simulator [23] was used. In addition to simulated reads, ART generates a SAM alignment file for error-free versions of the simulated reads. This alignment file can be used to extract the error-free reads as well as the read locations within the given reference genome.

The evaluation on real-world data is more challenging since error-free versions of the data are typically not available. Instead of per-nucleotide checks, k-mer analysis and de-novo assembly are performed for real-world datasets. Our k-mer

analysis gives information about how well the data in the corrected data matches a reference genome by comparing the k-mers of a dataset to those of the respective genome. In de-novo assembly, a new genome is assembled from the corrected data and compared the a reference genome.

The evaluation results of CARE are compared to the state-of-the-art tools Musket v1.1, SGA v0.10.15, Karect (Github commit from 16th March 2015), Bcool (Github commit from 29th November 2018), Lighter v1.1.2, and BFC r181. All tools, including CARE, come with multiple program settings that affect the results, and usually there is no perfect setting that always produces the best possible result for all datasets. Default settings or recommended settings are used for the other tools. For CARE, fixed settings are used during comparison with other tools. However, we also give an overview how different settings impact the results of CARE. This is done by selecting a subset of options that are varied independently.

In the remainder of this chapter, datasets are introduced which are used for evaluation and it is explained how Random Forests are trained for use with CARE. Then, the results are presented starting with the evaluation of different program settings, followed by the comparison of results to other tools on simulated data and real-world data. A detailed runtime analysis is given in chapter 6. It compares the runtime between all tools on a large dataset with 900M reads, and performs a detailed comparison of the parallelization strategies employed by CARE.

5.1 Datasets

Both simulated datasets and real-world datasets were used, which are presented in Tables 5.1 and 5.2. There are four different collections of simulated datasets (A-D), and a collection of real datasets (R). All used datasets come from four different reference genomes: Drosophila melanogaster (D.melanogaster), Caenorhabditis elegans (C.elegans), full Human, and only Human Chromosome 14. Collections A and B were generated using ART's built-in HiSeq 2000 sequencing profile. This produces paired-end reads with an error-rate of approximately one percent and a read length of 100. Collections C and D use the MiSeqV3 profile that generates paired-end reads of length 250 with approximately two percent error-rate. Datasets in collection A and C (B and D) have a coverage of 30x (60x). Datasets R1 and R2 are available from the Short Read Archive using the accession numbers SRR543736

and SRR988075, respectively. Dataset R3 comes from the Genome Assembly Gold-standard Evaluations (GAGE)[1].

Number	Organism	Reads (A)	Reads (B)	Reads (C)	Reads (D)
1	C.elegans	30.1M	60.2M	12.0M	24.1M
2	D.melanogaster	36.0M	72.1M	14.4M	28.9M
3	Hum. Chr. 14	26.5M	53.0M	10.6M	21.2M
4	Human	914.7M	-	-	-

Tab. 5.1.: Simulated datasets of collections A-D with their respective number of reads. Dataset number 4 is only available in collection A.

Number	Organism	Coverage	Reads
1	C.elegans	58x	57.7M
2	D.melanogaster	64x	75.9M
3	Hum. Chr. 14	35x	36.5M

Tab. 5.2.: Collection R of real-world HiSeq datasets.

5.2 Training of Random Forests

Multiple Random Forests were trained using simulated HiSeq datasets with 30x coverage from different organisms, namely C.elegans, D.melanogaster, Mus musculus Chr. 15, Human (Chr. 14 and Chr. 15), and A.thaliana. First, the CPU version of CARE was used to extract the features for each dataset for anchor correction and candidate correction, respectively. Features were subsequently labeled by comparing the consensus nucleotide to the corresponding position in the error-free version of the read. If they are equal, the feature is labeled *true*, meaning the consensus nucleotide is correct and should thus replace the original nucleotide in the erroneous read. Else, the feature is labeled *false*.

Five Random Forests with 128 trees were then trained for both anchor correction and candidate correction, respectively. To reduce the chance overfitting in the evaluation, the forests were trained following a leave-one-out approach. For each forest, features from a different organism were excluded from the training. Evaluation on the datasets was then performed using the Random Forest which did not include training data from the organism whose reads should be corrected. The resulting decision trees have an average depth of around 51 and 56 for anchor correction and candidate correction, respectively.

[1] `https://gage.cbcb.umd.edu/data/index.html`

5.3 Variation of program settings

CARE has many program parameters which can affect correction quality. This section aims to highlight the impact of a limited number of parameters on dataset A3. The following parameters are used as a baseline, which are then modified individually.

Quality scores	Enabled, 8-bit
Kmer size	20
Hash functions	48
Paired-end mode	yes, t_paired = 0.06
Candidate corrections	Enabled
Random Forest	disabled

With the baseline settings, CARE achieves the following correction statistics.

TP	$23,558,011$
FP	$10,646$
FN	$2,093,616$
TN	$2,622,999,127$

Figure 5.1 shows TP and FP depending on k-mer size, number of hash functions, quality score configuration, and pair-mode. Those parameters influence the constructed MSAs by altering the set of candidate reads and their weights within the MSA. Kmer size varies from 10 to 32. The number of hash functions varies from 1 to 48. The threshold for pair-mode increases between 0.01 and 0.20 in steps of 0.01. Note that the results are unaffected from this threshold if the input file is treated as single-end (SE) instead of paired-end (PE).

The smaller the k-mer size the more likely it is to find a specific k-mer in a read. The average number of assigned reads per hash value increases. During the hash table compaction step, more keys i.e. hash values will be removed because the bucket size is exceeded more often. Both observations alter the initial set of candidate reads. Reads of the same genomic region as the anchor may no longer be a candidate, because the frequent k-mers have been removed. Additionally, the set of candidates may be larger, because more reads share a k-mer. This increases the runtime since more pair-wise alignments need to be computed. On the other hand, increasing the k-mer size will decrease the candidate set size. The probability that two reads share a large k-mer decreases. Additionally, a single sequencing error will produce k erroneous k-mers, which may not be found in the dataset. With a k-mer size of 16

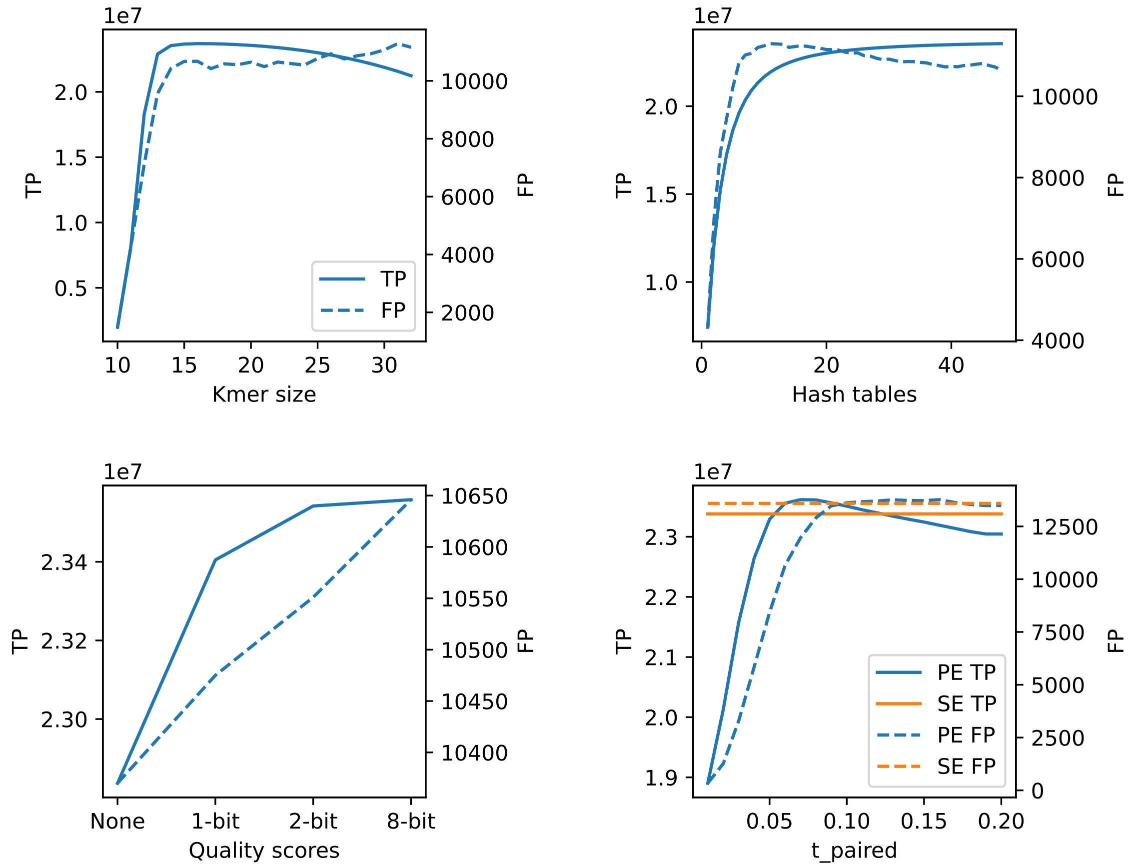


Fig. 5.1.: Variation of correction parameters.

or less it is possible to reduce the memory usage of hash tables because the k-mers and their hash values can be represented by 32-bit integers, whereas larger k-mers require the use of 64-bit integers instead.

When an erroneous k-mer produces a minimal hash value, i.e. its hash value contributes to the read signature S, it is less likely to find reads from the same genomic region with that same hash value for the same hash function. Thus, a larger number of hash functions has two benefits. First, it increases the probability to use non-erroneous k-mers in the signature. Second, when more hash functions are used, the corresponding k-mers of the signature in general cover more positions of the read. This increases the chance to find candidate reads which cover different parts of a read. Since different candidates may overlap different positions of a read, it is important to have each position of an anchor read covered by a candidate to be able to construct a high quality MSA. The drawback of using many hash functions is the high memory usage which scales linearly with the hash functions / hash

tables. Additionally, runtime increases because more tables need to be accessed, and candidate sets are typically larger.

Quality scores correspond to the probability that a specific nucleotide is called correctly by the sequencing machine. Recall that the consensus of an MSA column is determined by the nucleotide weights. The use of quality scores improves the accuracy of the MSA consensus since positions of low quality contribute less weight. This is true for even the simplest approach of one 1 bit positions which classifies a nucleotide as either "good" or "bad". With a higher resolution for quality scores, fine-grained weight contributions are possible. The usage of quality scores is always recommended if memory permits.

In single-end mode, CARE achieves $23,380,267$ TP and $13,589$ FP. Recall that in paired-end mode, candidate pairs which span across anchor pairs are always kept in the candidate set during alignment filtering. Candidates for which no such pair exists are only kept if the relative number of mismatches in the alignment overlap does not exceed t_paired. The smaller t_paired, the better the accepted alignments. However, at the same time, the number of accepted alignments decreases. This can lead to MSAs with low coverage that may in turn be unable to correct an error. With its default setting of 0.06, CARE achieves both a better number of TP and FP in paired-end mode compared to the single-end mode.

Next, we take a look at candidate correction and random forest usage. Instead of directly affecting the MSAs, these options change the way how the constructed multiple-sequence alignments are used to obtain the final corrected read. There are two different types of anchor corrections. Those constructed from a high-quality (HQ) MSA, and those coming from a low-quality (LQ) MSA. As described in Section 4.3.3, high-quality corrections are always accepted as final correction, whereas low-quality corrections may be either accepted or rejected depending on the presence of candidate corrections. The impact of candidate corrections is two-fold. Wrong low-quality corrections can be avoided which reduces the number of false-positives. At the same time, however, valid corrections could be rejected because of the lack of supporting candidate corrections. This manifests as a reduction of true-positives. Random forests allow for a more precise correction of low-quality anchors, and candidates, which improves correction quality in terms of both TP and FP.

Figure 5.2 and Figure 5.3 show the contributions to TPs and FPs, respectively, of the three different correction types in different settings. The three types are HQ anchor correction, LQ anchor correction with supporting candidates (LQ+good cands), and LQ anchor correction without supporting candidates (LQ+bad cands). One can clearly see that low-quality corrections without candidate support are responsible for

the majority of wrong nucleotide corrections. Note that the displayed contribution of high quality anchors is the same across all selected settings since they are not affected by either Random Forest corrections or candidate corrections.

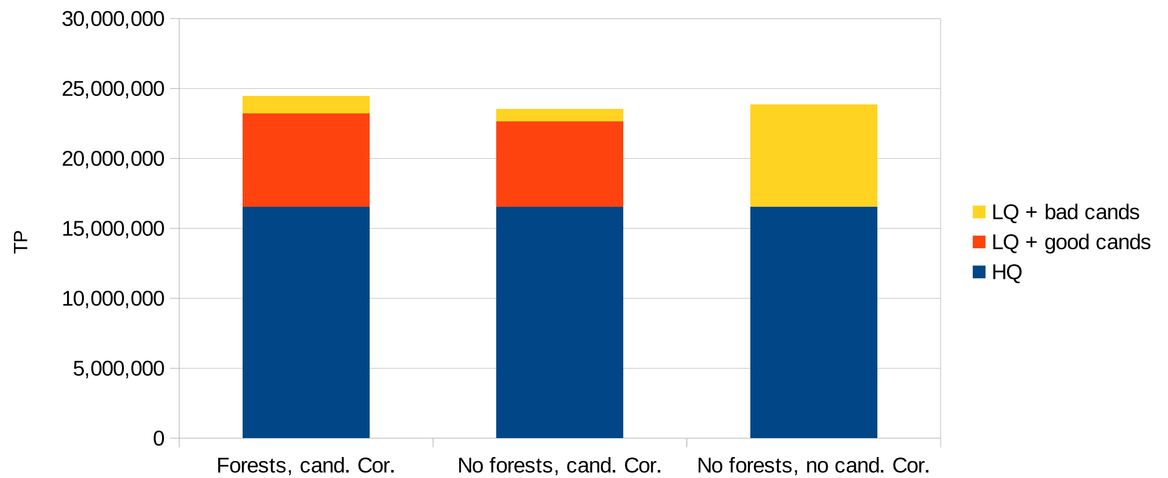


Fig. 5.2.: TP depending on random forest usage and candidate correction.

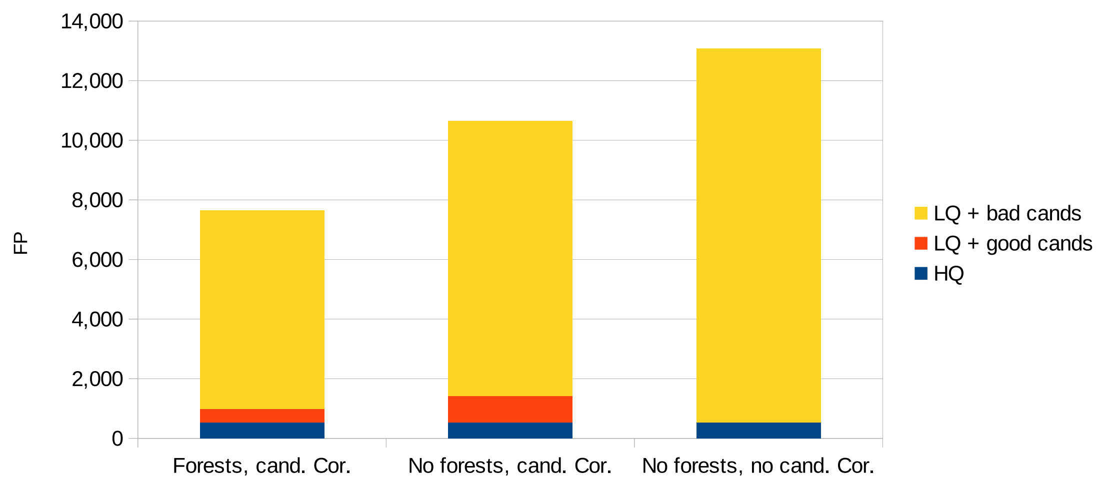


Fig. 5.3.: FP depending on random forest usage and candidate correction.

5.4 Evaluation on simulated HiSeq datasets

CARE was run with the following settings. All remaining tools were run with their recommended settings or default settings.

CARE settings	Dataset A(B)1-3	Dataset A4
Quality scores	Enabled, 8-bit	Enabled, 2-bit
Kmer size	20	
Hash functions	48	
Paired-end mode	yes, t_paired $= 0.06$	
Candidate corrections	Enabled	
Random Forest	Enabled, 128 trees, thresholds $93; 15$	

Because of memory constraints, full 8-bit quality score could not be used for dataset A4. Instead, quality scores were stored in the compressed 2-bit format.

For all datasets the nucleotide statistics per tool were determined as well as the derived metrics sensitivity, specificity, precision, and false positives per million corrections. Derived metrics are defined as follows:

- sensitivity $= \frac{TP}{TP+FN}$
- specificity $= \frac{TN}{TN+FP}$
- precision $= \frac{TP}{TP+FP}$
- false positive rate (FPR) $= 1,000,000 \cdot \frac{FP}{TP+FP}$

The false positive rate is better suited to highlight differences in false positive corrections than the other derived metrics. This is because in general the absolute numbers of true positives and true negatives are multiple orders-of-magnitude greater than those of false positives. Thus, even large changes in false positive numbers may not accurately be accounted for by specificity and precision.

Figure 5.4 shows the average ratio of TP and FP per tool over CARE+RF. This is computed as $\frac{TP_{TOOL}}{TP_{CARE+RF}}$ and $\frac{FP_{TOOL}}{FP_{CARE+RF}}$, averaged over all datasets per collection. All tools achieve similar TPs. In contrast, the number of false-positive corrections varies significantly per tool. CARE produces up to two orders-of-magnitude fewer false positive corrections than those of the competitors which makes it superior. For dataset collection B with 60x coverage, CARE with enabled random forest produces less TP on average than without forests. This could be because the forests have been trained with a different coverage. However, while the training data may contribute to this observation, the root cause may be different since in terms of FPs the distance to other tools decreases. Still, CARE produces the most accurate corrections. Tables with detailed results can be found in the appendix, Section A.1.

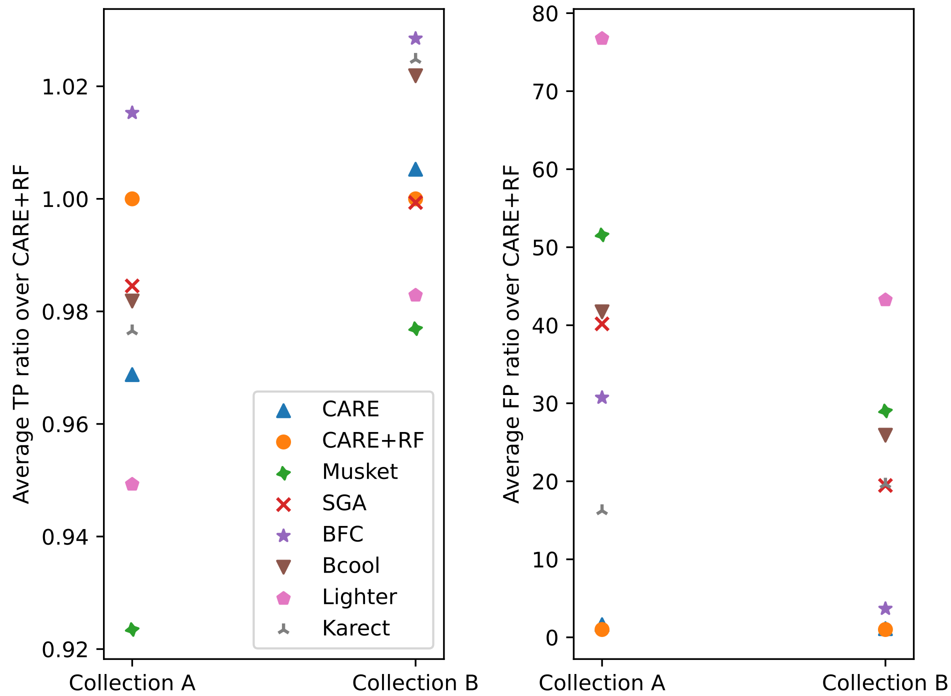


Fig. 5.4.: Average TP ratio and FP ratio over CARE RF on simulated HiSeq data. Greater numbers are better for TP. Smaller numbers are better for FP.

5.5 Evaluation on simulated MiSeq datasets

All tools, including CARE, were not specifically designed for MiSeq data. It differs from HiSeq data in both read length and error rate. Above evaluation was repeated on dataset collections C and D to see how the error correctors can handle such data. As before we computed the average ratios, which are shown in Figure 5.5.

Generally, the field lies closer together in terms of FP compared to the HiSeq case with the exception of Musket which struggles with the different type of data. CARE produces at least 7 times less FP on average compared to other tools. For TPs there is a greater variation of rations. Similar to HiSeq correction, using the random forest produces less TP than without random forest with a dataset coverage of 60x.

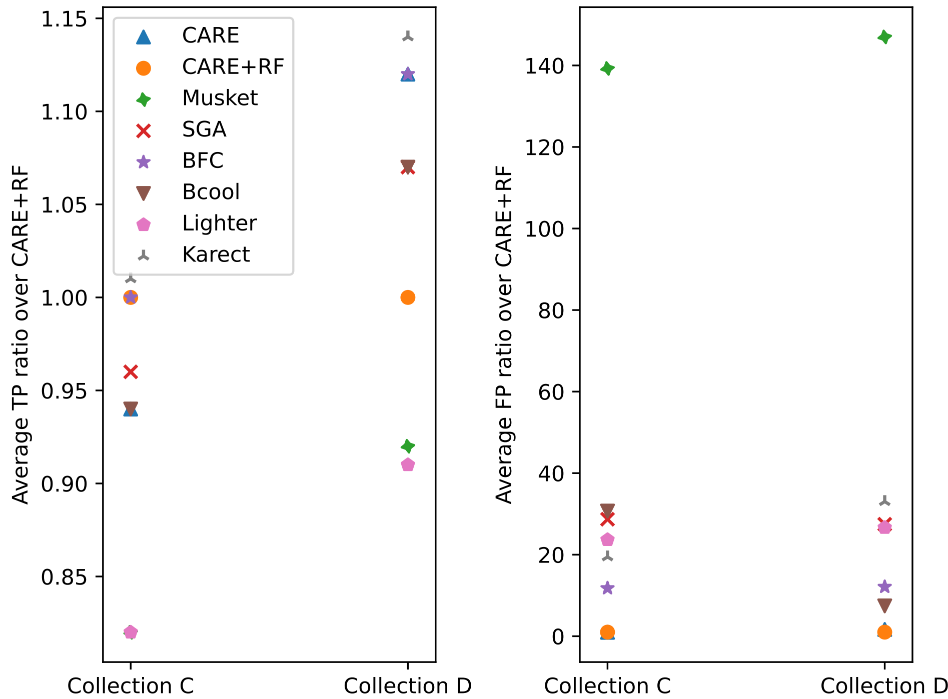


Fig. 5.5.: Average TP ratio and FP ratio over CARE RF on simulated MiSeq data. Greater numbers are better for TP. Smaller numbers are better for FP.

5.6 Evaluation on real-world datasets

This section focuses on the applicability of error correction on real-world data. In contrast to simulated datasets, the location of sequencing errors in real-world datasets is generally unknown which prohibits the calculation of per-nucleotide statistics. Instead, dataset analysis is performed on both uncorrected datasets and their corresponding corrected datasets to see if the corrected dataset produces better results. The real-world evaluation focuses on two common use-cases: k-mer analysis and de-novo assembly.

5.6.1 K-mer evaluation

The impact of wrong corrections on the k-mers is two-fold. On one hand, a valid k-mer could be changed into one which does not appear in the reference genome. On the other hand, k-mers of reads spanning low-coverage regions could be altered into more frequent ones. While this does not introduce wrong k-mers, correct information is lost. To quantify false-positive corrections on real-world datasets

where the location of errors is generally unknown, k-mer spectra produced by Jellyfish v2.3.0 [53] were inspected for uncorrected reads, corrected reads, and the corresponding reference genome. A k-mer is called true k-mer if it occurs in the reference genome. During error correction true k-mers which are present in the uncorrected reads should not be altered to keep the correct genome information. k-mers which are present in both the uncorrected reads and the genome, but are missing from the corrected reads are called lost true k-mers. A perfect error correction algorithm should not introduce lost true k-mers.

We computed the number of low-coverage (≤ 10 coverage) lost true 21-mers of datasets R1,R2 and R3. Low-coverage k-mers can be easily lost during error correction because there is not much supporting information. Table 5.3 shows the results for CARE, and for the three best competitors, on dataset R1. Full results for R1, R2, and R3, are available in the appendix Section A.2. The results show that datasets corrected by CARE have the fewest numbers of removed low-coverage true 21-mers. This indicates accurate corrections with rare false-positives.

Coverage	CARE	CARE+RF	BFC	SGA	BCOOL
1	3,700	2,888	11,496	5,567	6,675
2	328	247	4,710	1,309	3,045
3	73	39	2,928	135	1,902
4	59	9	845	25	1,449
5	14	2	172	3	1,368
6	10	1	41	3	1,387
7	21	0	29	0	1,237
8	4	0	4	2	1,307
9	11	0	1	0	1,447
10	21	0	1	0	1,686
Sum	4,241	3,186	20,227	7,044	21,503

Tab. 5.3.: Lost 21-mers for dataset R1.

5.6.2 De-novo assembly evaluation

In de-novo assembly, a collection of reads is used to reconstruct the genome of the individual whose cell samples were sequenced to obtain the reads. This is achieved by identifying overlapping reads and joining them into longer sequences, so called contigs. Ideally, the assembly process produces only long, error-free contigs which accurately represent portions of the original genome. Subsequently, the contigs will be arranged into longer scaffolds. The presence of sequencing errors and false-positive corrections can reduce the assembly quality of contigs. For example,

sequencing errors can lead to dead ends when no similar overlapping read can be found to elongate the contig. This results in a more fragmented assembly, i.e. shorter contigs. Conversely, unrelated reads could be joined in a contig because sequencing errors make them appear similar.

Assembly was performed with SPAdes v3.13.1 [4]. The integrated error correction step of SPAdes was disabled. Assembled contigs were analysed using QUAST v5.0.2 [18].

The evaluated metrics are: the number of contigs longer than $50,000$ basepairs, the accumulated length of those contigs, the total number of contigs, the N50 score and NG50 score, and the number of misassembled contigs. The N50 value is defined as follows:

"The N50 of an assembly is a weighted median of the lengths of the sequences it contains, equal to the length of the longest sequence s, such that the sum of the lengths of sequences greater than or equal in length to s is greater than or equal to half the length of the genome being assembled. As the length of the genome being assembled is generally unknown, the normal approximation is to use the total length of all of the sequences in an assembly as a proxy for the denominator." [14]

The NG50 differs from the N50 value in that the actual size of a reference genome is used instead of the total length of the assembly.

Table 5.4, shows an excerpt of the analyses for datasets R3 for CARE and the three best competitors. Appendix tables A.8, A.9, and A.10 show excerpts of the analyses for datasets R1, R2, and R3, for all tools.

On dataset R1, only SGA and CARE are able to increase the N50 value compared to the assembly generated from the original unprocessed data. CARE achieves the fewest misassembled contigs. Assembly results for R2 improve when corrected with either BCOOL and CARE. BCOOL produces the fewest and longest contigs. All tools are able to produce better results for R3. Contigs generated after correction with CARE are the longest. Their numbers are the fewest.

R3	Uncorrected	CARE	CARE+RF	BFC	Musket	Karect
# contigs $\geq 50k$	3	51	63	57	52	53
# contigs	18,340	10,941	10,635	10,697	11,211	10,820
# misassembled	135	611	554	603	734	577
N50	7,859	14,196	14,749	14,673	13,742	14,436
NG50	5,506	10,251	10,624	10,470	9,886	10,263

Tab. 5.4.: Selected assembly results for dataset R3.

6 Performance of CARE

In this chapter, we take a closer look on the parallelization of CARE. The program comes with different strategies to leverage the capabilities of modern hardware to achieve good performance. On one hand, multi-threading targets multi-core CPUs. The more interesting topic is the performance on many-core GPUs. As will be seen, the GPU version of CARE greatly outperforms the purely CPU-based implementation, but comes with its own challenges to achieve high performance. Aside from the performance of the different modes, this chapter also includes a performance comparison between different error correction tools on the large Human dataset (A4).

The following systems were used for our performance benchmarks:

M1 (Single-GPU workstation): AMD Ryzen Threadripper 3990X 64-core CPU, 256 GB DDR4 RAM, NVIDIA A100 PCIe GPU with 80 GB HBM2e memory, CUDA Toolkit 11.6

M2 (Multi-GPU server): Dual-socket AMD EPYC 7713P 64-core CPU, 2,048 GB DDR4 RAM, 8 fully-connected A100 SXM4 GPUs with 80 GB HBM2e memory, CUDA Toolkit 12.1

We begin with benchmarks for dataset A1 (C.elegans, 30.1M reads, Length 100, HiSeq) on system M1. Unless mentioned otherwise, the following settings were used. The batchsize for the GPU version was set to 4096.

Quality scores	Enabled, 8-bit
Kmer size	20
Hash functions	48
Paired-end mode	yes, t_paired $= 0.06$
Candidate corrections	Enabled
Random Forest	Disabled

6.1 Construction phase

The construction of the readstorage involves parsing the input files to obtain the raw sequences and quality scores, and converting the sequences into the 2-bit representation. Optionally, quality scores are encoded, as well. First, we take a look at the maximum achievable performance of our custom file parser in isolation. The performance depends on the file type (FASTA, FASTQ, compressed (.gz) or plain-text) and on the storage type (HDD, SSD, RAM). For this experiment, we converted dataset A1 to all four different file types and measured the required time to count the number of reads in the file. The results are presented in Table 6.1.

File type File size	fasta 3.6 GB	fasta.gz 0.98 GB	fastq 6.5 GB	fastq.gz 2.8 GB
RAM	1.07	11.52	1.84	30.68
SSD (PCIe)	1.59	11.89	2.70	31.56
HDD	7.03	12.04	12.64	31.94

Tab. 6.1.: Runtime in seconds to count the number of reads per file.

Most of the runtime for compressed files is spent decompressing the data. As expected, the decompression of a gzip file is several times slower than the processing of the plain-text counterpart. Since most of the time is spent on decompression, faster storage access has negligible impact in this case.

Recall that a multi-threaded three-stage pipeline is employed to construct a readstorage. Multiple threads are used to encode sequences into 2-bit format. Figure 6.2 shows the total pipeline runtime depending on the number of threads utilized for encoding.

Threads	fasta	fasta.gz	fastq	fastq.gz
1	18.85	18.86	18.94	34.7
2	9.49	15.61	9.55	34.68
4	5.16	15.53	6.68	34.41
8	5.13	15.54	6.79	34.35

Tab. 6.2.: Scaling of readstorage construction with the number of encoder-threads. Runtime is given in seconds. The input file is already cached in RAM.

Sequence encoding scales up to 4 threads for plain-text inputs. Additional threads cannot be utilized because file parsing becomes the limiting bottleneck. Note that the full pipeline introduces additional overheads such as data copies and communication compared to stand-alone file parsing. Similar to the previous experiment, the most compute-intensive part of processing compressed inputs is decompression. In this case, sequence encoding takes only a small fraction of the total runtime.

Next, we take a look at minhasher construction. The generation of key-value pairs from reads are trivially parallelizable. On the other hand, the compaction step involves more elaborate algorithms such as sorting and stream compaction which are not trivially parallelizable and which have a data-dependent performance. Figure 6.1 shows the speed-up of a parallelized minhasher construction over the single-threaded CPU version. The single-threaded CPU runtime is $276s$. Host-side multi-threading in the GPU version only has limited impact as most of the computations are performed on the GPU instead of the CPU. The benefit of using more than a single thread in the GPU version with CPU tables is overlapping compaction and key-processing. The compaction of the key-value pairs of the next table can be performed in a different thread at the same time as compacted keys of the current table are inserted into the single-value hash table. The construction of GPU-accelerated hash tables is the fastest.

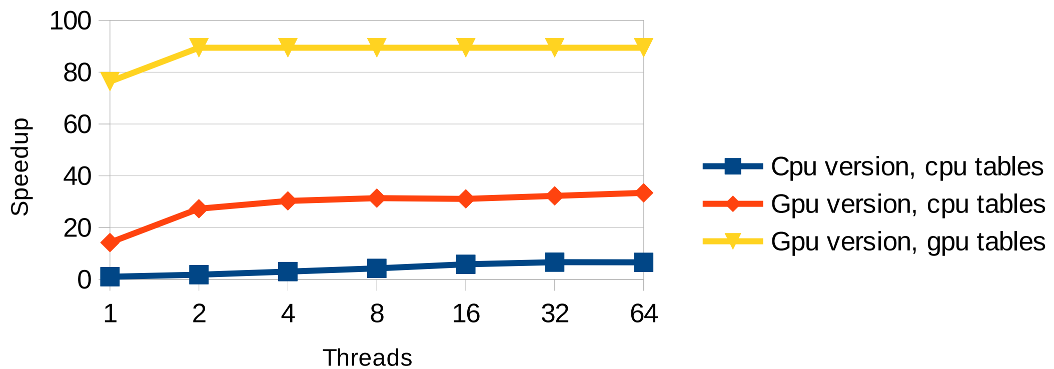


Fig. 6.1.: Speed-up of hash table construction.

6.2 Correction phase

Recall that the correction phase comprises of multiple steps. Figure 6.2 shows the relative time spent on these steps for the CPU version and the GPU version with host-sided hash tables, and device-sided hash tables, respectively. One can easily see that hash table accesses in host memory severely limit the performance of the GPU version. This major contribution to runtime explains the need for additional multi-threading to parallelize those hash table queries.

Out of the total execution time of around $6050s$ for the single-threaded CPU version, $5735s$ (95%) are spent in the correction phase. Since reads can be processed independently, this phase can be easily parallelized on the CPU with multi-threading.

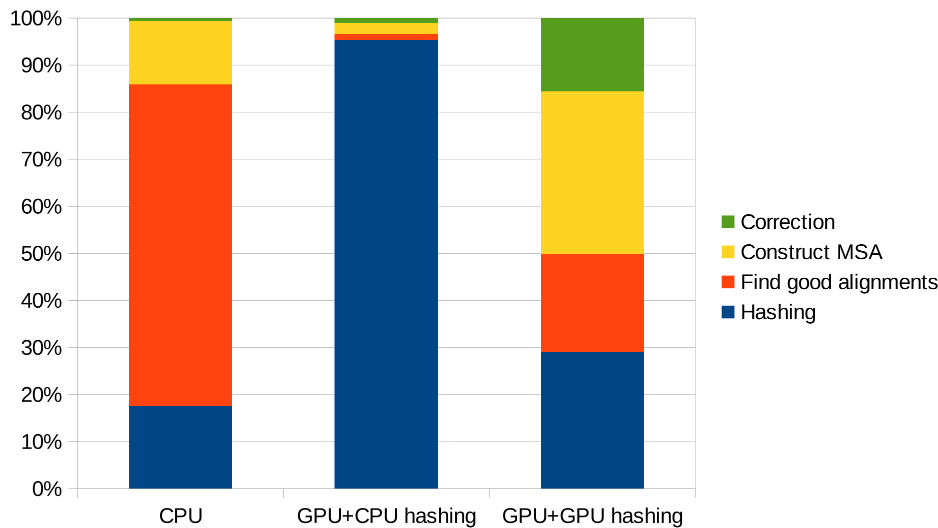


Fig. 6.2.: Relative time spent in the different steps during correction phase.

For the GPU version, however, multiple parallelization schemes exists. Similar to the CPU version, the simplest approach of the GPU version (GPU simple) is to use a host-side parallel for-loop over the reads where each thread executes all steps for the reads of its current batch on the device. This approach is applicable for both types of hash tables. To tackle the high workload of CPU hash tables in the GPU version, a producer-consumer pipeline can be used instead of the simple approach for more efficient resource utilization. Its performance depends on the numbers of producers and consumers. For the presented benchmarks, the number of consumers can be either one (GPU pipeline 1), or two (GPU pipeline 2). For the pipelined approaches, the number of producer threads is given by the total number of threads minus the number of consumer threads. In general, more consumers can be selected. However, this is often unfeasible because producers cannot deliver hash table results fast enough to saturate more than two consumers. Figure 6.3 shows the speed-up in the correction phase over the single-threaded CPU version.

With 64 threads, the CPU version achieves a speed-up of 49 over the single-threaded implementation. Considering the GPU approaches with host-sided hash table operations, the simple version performs best if the number of used threads is low. This is because the approaches with a producer-consumer pattern do not use all available threads for producers, i.e. hash table queries. Increasing the number of threads in the simple version has two disadvantages. First, concurrent CUDA commands, for example launching kernels or performing memory operations, will eventually be serialized by the driver which introduces contention and limits host-side throughput. Second, each thread needs its own working set. The most amount of device-memory

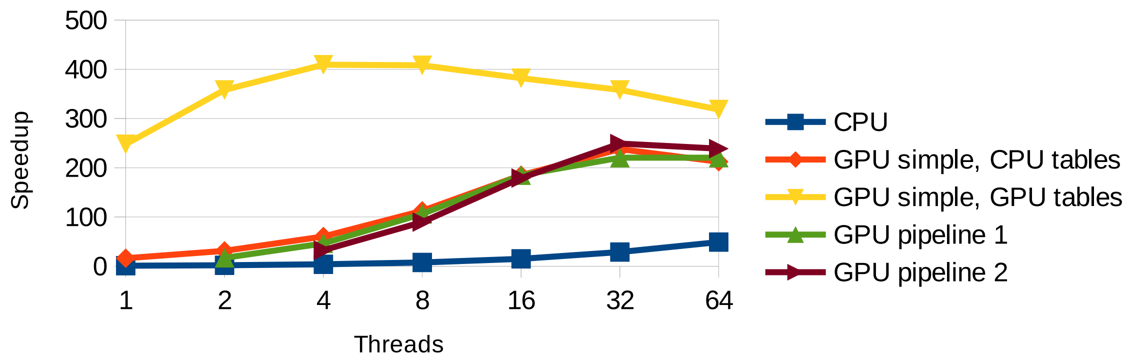


Fig. 6.3.: Speed-up over the single-threaded CPU version in the correction phase.

is required for the computations of alignments and multiple-sequence alignments. Both drawbacks are accounted for when using the producer-consumer approach. The number of consumers, which issue most of the GPU work, is limited to at most two. This in turn reduces the driver contention as well as the number of large working sets.

Switching to GPU-based hash tables removes the main performance bottleneck. With only four threads, the warpcore-based approach with GPU hash tables achieves the best performance. It is the most efficient parallelization. However, this comes at the expense of high memory usage to store the hash tables in GPU memory. The GPU memory consumption in megabyte is reported in Table 6.3.

Threads	1	2	4	8	16	32	64
GPU simple, CPU tables	4,307	4,405	4,601	4,964	5,687	7,135	9,517
GPU pipeline 1	-	4,307	4,341	4,409	4,514	4,755	5,239
GPU pipeline 2	-	-	4,405	4,473	4,578	4,819	5,303
GPU simple, GPU tables	11,657	11,979	12,369	13,133	14,509	16,627	20,961

Tab. 6.3.: GPU memory usage [MB] during correction phase.

6.3 Merge phase

During the merge phase of the GPU version, corrections can be sorted on the GPU. Its performance is limited by the available device memory and transfer rate. Common GPU sorting algorithms such as radix sort do not operate in-place. Thus, both the unsorted data and sorted data need to fit into device-accessible memory. If the available memory is not sufficient, managed memory is used as a replacement which

allows the GPU to transparently access data residing on the host via PCIe bus. The used workstation provides enough device memory to sort the corrections of dataset S1 without the need of managed memory. This results in a speed-up of approx. $4.60s/0.62s = 7.42$ over the CPU-based sort, including data transfer times.

6.3.1 Overall performance

The total speed-up over the single-threaded CPU version is presented in Figure 6.4.

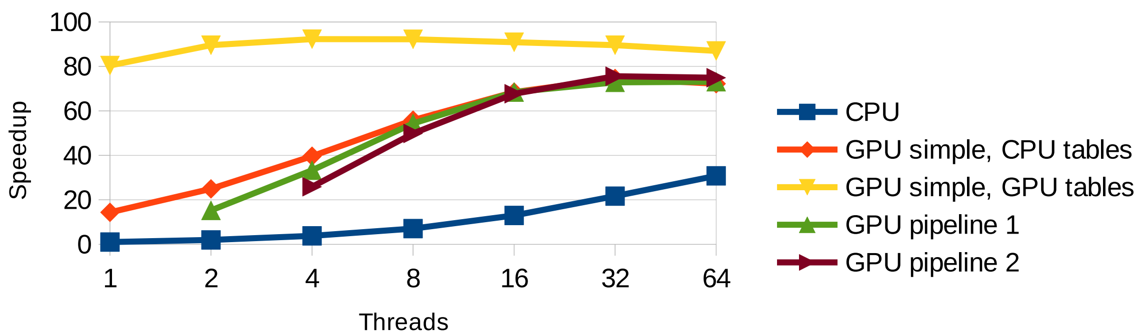


Fig. 6.4.: Total speed-up of GPU version over CPU version.

In the following, only approximate runtimes are mentioned. The single-threaded CPU version takes a total of 101 minutes to complete. The times spent in construction phase, correction phase, and merge phase are $280s$, 5730, and $30s$, respectively. With 64 threads, the total runtime is decreased to $200s$ $(50s + 120s + 30s)$ which corresponds to a speed-up of around 30.

The warpcore-based GPU approach achieves its peak performance with 4 CPU threads submitting work to the device. The total runtime is $70s$ $(25s + 20s + 25s)$, which is around 2.8 times faster than the CPU version with 64 threads. The total runtime of GPU pipeline 2 using 32 threads is $85s$ $(30s + 30s + 25s)$. The speed-up over the fastest CPU runtime is around 2.3.

Regarding overall memory usage, dataset S1 consists of $30,085,710$ reads of length 100. Including quality scores, reads occupy a total of $3,673$ MB in memory. The 48 hash tables use $6,400$ MB. The remaining free host memory holds anchor corrections and candidate correction. For dataset A1, those corrections add up to a total of $4,307$ MB.

6.4 Performance with random forests

The previous performance benchmarks were repeated with random forest correction. The corresponding results are shown in Figures 6.5, 6.6, and 6.7.

When random forests are used, the runtime increases. This is caused by irregular, effectively random, memory access patterns when traversing the trees. This is a performance hit especially for the GPU version, which requires contiguous memory accesses for fastest access times.

With enabled Random Forests, the time spent in the correction phase by the single-threaded CPU version increases by 33.5% from $5730s$ to $7650s$. In contrast, using forests with the GPU version comes with an increase in runtime of up to 55%. The actual factor depends on the used parallelization scheme and on the number of threads. The greater the GPU utilization without random forests, the larger the increase in runtime when they are used. For example, the performance of the simple GPU approach with CPU hash tables is limited by host-side hash table operations. Enabling the use of random forests only increases the time spent in the correction phase by up to 8%. The host-side performance is also limited by CUDA API serialization caused by concurrent API calls from different threads. This is most notable for high thread counts which submit GPU work.

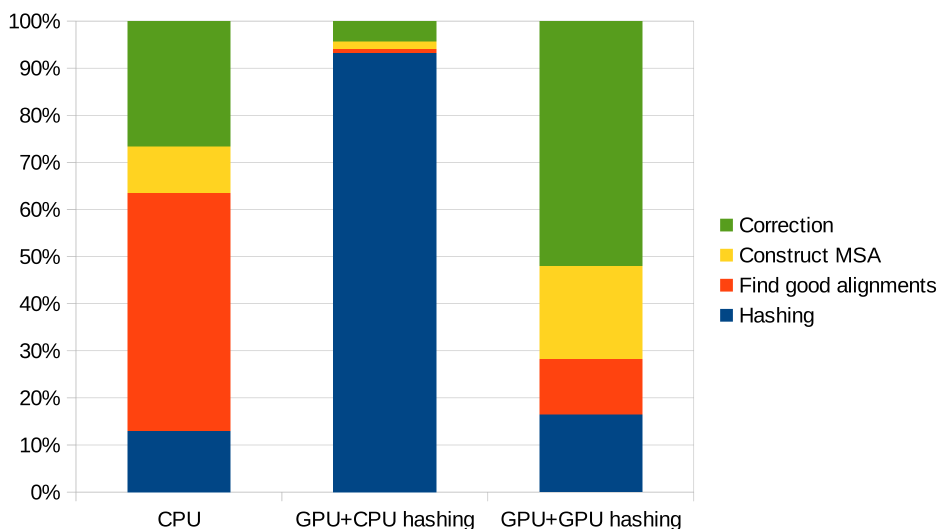


Fig. 6.5.: Relative time spent in the different steps during correction with random forests.

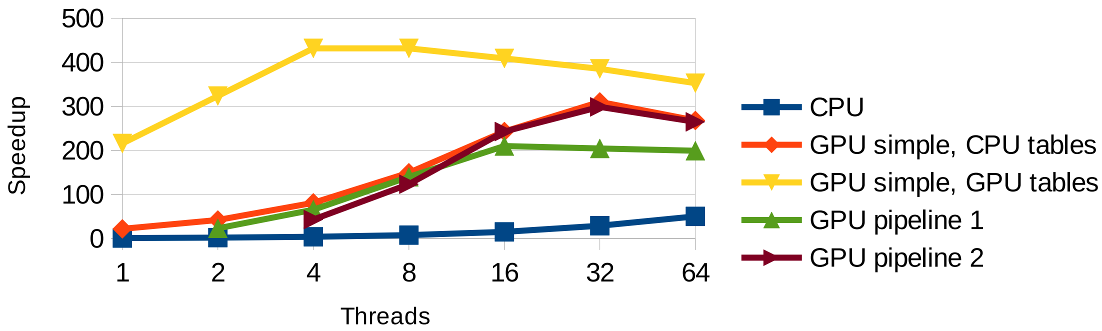


Fig. 6.6.: Speed-up of GPU version over CPU version in correction phase with random forests.

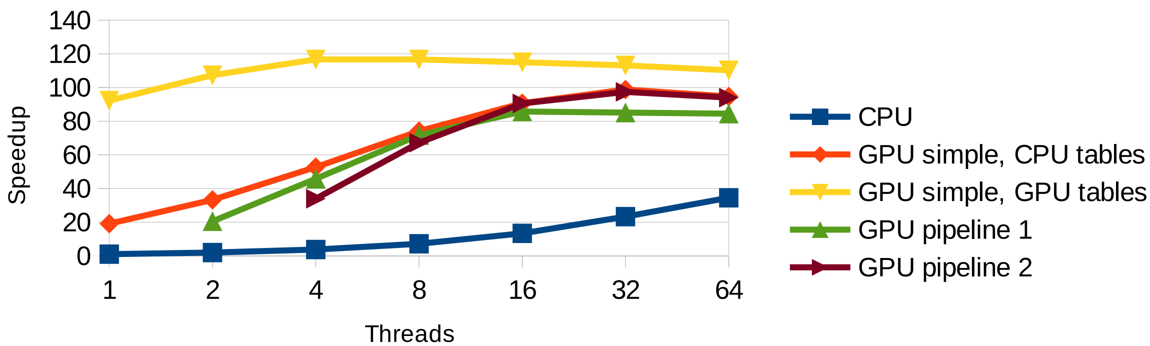


Fig. 6.7.: Total speed-up of GPU version over CPU version with random forests.

6.5 Read placement

For best performance, all data should be located on the GPU. Previous results focused on the location of hash tables. However, the location of reads in the read-storage is important, as well. If enough device memory is available after hash table construction, the complete set of reads can be transferred to the GPU. This is the case for the previous results. In this section, we take a look at the performance of the correction step for the case when read data does not fit completely on the GPU and must thus be distributed between host and device. This may be the case for large datasets. In principle, there are three possibilities to distribute the reads between the CPU memory and GPU memory: 1. Sequences on host, quality scores on host. 2. Sequences on device, quality scores on host. 3. Sequences on host, quality scores on device. Of course, it may also happen that either the array of sequences or the array of qualities must be distributed between the memory spaces. However, we do not

consider this case here. The performance simply improves with the amount of data stored in GPU memory.

The best performance in the previous benchmarks was achieved with GPU hash tables. Thus, moving data from the device to the host should have the greatest impact which makes using GPU tables the most suitable configuration for this benchmark. Figure 6.8 shows the runtimes of the correction phase with GPU hash tables for the different read distributions in comparison to the full device data. Indeed, using one thread the runtime increases by a factor of 6 when all read data is located on the host. Using more threads, this problem is alleviated but the peak performance is unmet. Comparing the configurations where exactly one part resides on the GPU, the performance is better if quality scores are accessed on the device. This is because 8-bit quality scores occupy more memory than 2-bit sequences which would require more data to be sent over slow PCIe bus otherwise. Of course, this is also true if both parts are located on the host. This is confirmed by Figure 6.9 which shows the same experiment conducted with 2-bit quality scores. Here, the runtimes with either CPU sequences or CPU qualities are closer together. Placing qualities on the CPU is slightly faster because they can be accessed on the host concurrently to the alignment kernels. On the other hand, accessing sequences on the host cannot overlap with other work.

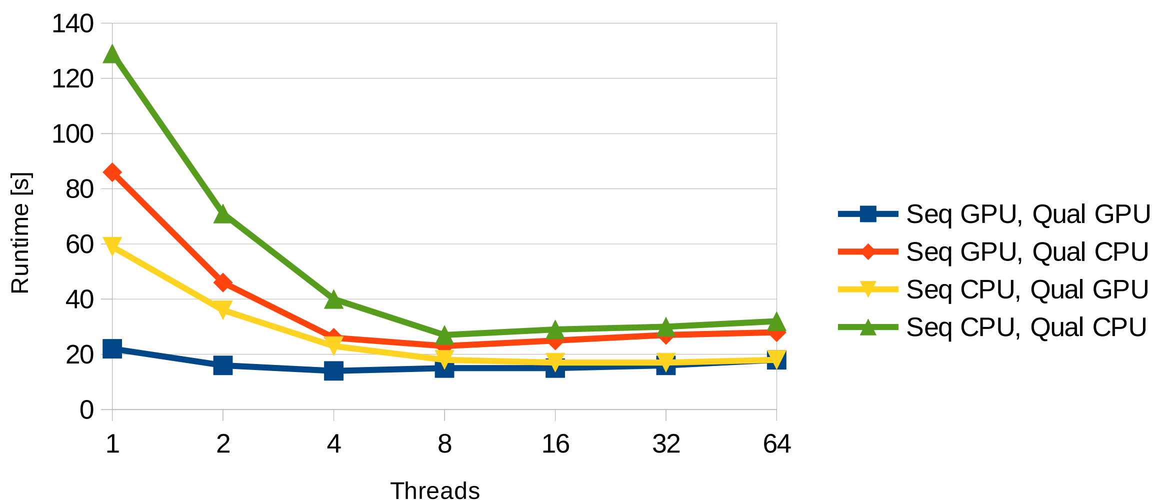


Fig. 6.8.: Runtime of the correction phase with GPU hash tables depending on read data location. 8-bit quality scores are used.

With common settings it is more likely that hash tables do not fit into device memory and have to be kept in host memory. In this case, the best performance is achieved with producer-consumer pipeline 2. Figures 6.10 and 6.11 show the data-dependent

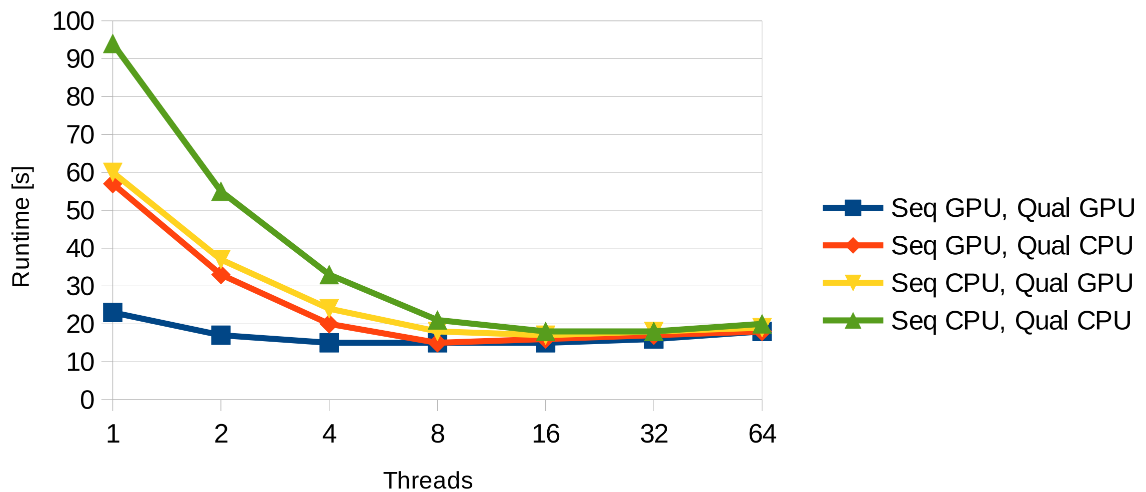


Fig. 6.9.: Runtime of the correction phase with GPU hash tables depending on read data location. 2-bit quality scores are used.

runtimes for this approach. Runtime behaviour is different from the version with GPU hash tables. This is because sequences and quality scores are accessed by different types of threads, respectively. Producer threads access the sequences, whereas the quality scores are accessed by the two consumer threads. For small numbers of producers, the GPU is underutilized and performance is limited by the processing speed of producers. The producers operate faster when sequences reside in device memory. On the other hand, if the number of producers is sufficiently high, the bottleneck shifts to the consumer threads which drive the main GPU computations. Those consumers benefit from quality scores located on the device.

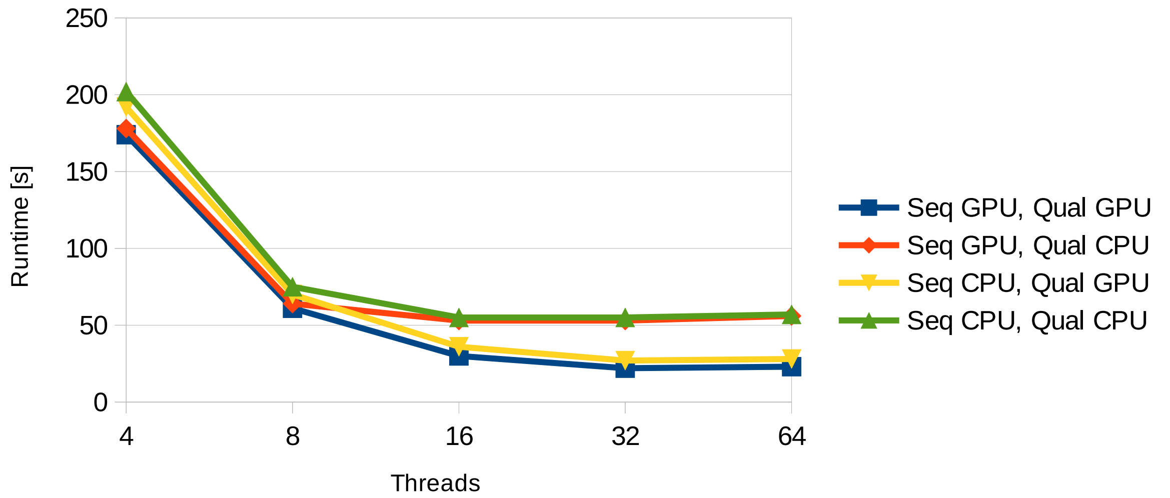


Fig. 6.10.: Runtime of the correction phase with CPU hash tables depending on read data location. 8-bit quality scores are used.

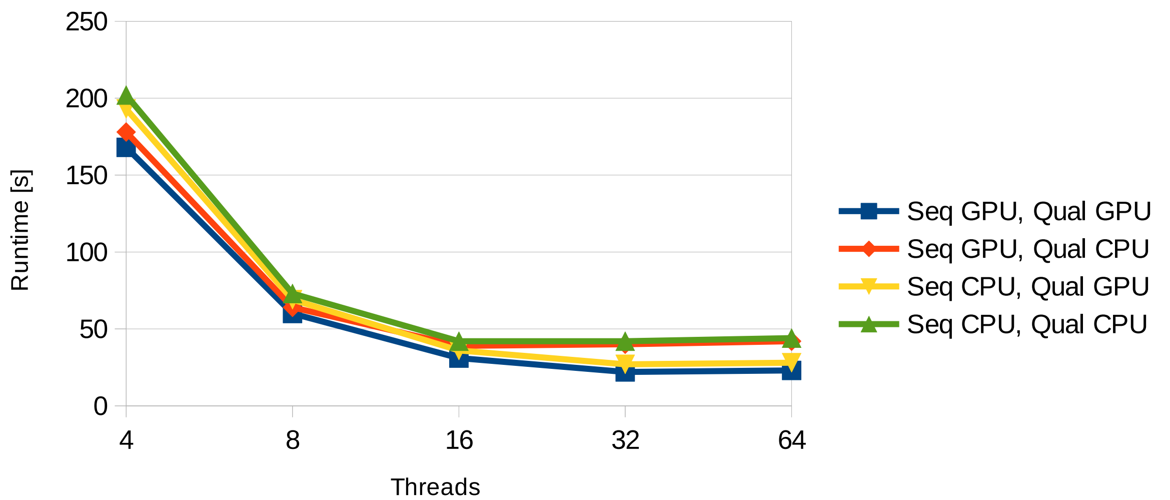


Fig. 6.11.: Runtime of the correction phase with CPU hash tables depending on read data location. 2-bit quality scores are used.

6.6 Performance comparison to other tools

To compare CARE to the state-of-the-art tools, performance was measured for the correction of the largest dataset, dataset A4 (Human, 900M reads, HiSeq) with machine M1. This dataset tests the multi-threading capabilities of the used tools, which are required for the correction in a reasonable amount of time. To achieve good performance, however, file access speed may be important as well. Two sets of benchmarks were performed where the tools either read from and write to an ordinary HDD connected via SATA, or read from and write to a two terabyte SSD connected via PCIe.

For the benchmarks, CARE used 2-bit quality scores to limit the memory consumption of reads. This allows to fit the whole read dataset comprising 47.7 GB into the device memory of the A100 GPU. 48 host-side hash tables are used which occupy 163.6 GB in CPU memory. GPU pipeline 2 is used for parallelization with two consumer threads. Table 6.4 presents the runtime and memory usage of the programs. Please note, however, that the performance values for CARE were obtained from an older version which did not include improvements implemented at a later point in time when system M1 has no longer been available. Those improvements are explained and evaluated in the next section. Nonetheless, we decided to present the collected data. The reasons are two-fold. It still allows a general comparison of execution times, and shows the possible performance benefits of using faster file storage.

Tool	Threads	Time HDD	Time PCIe SSD	Memory [GB]
CARE (CPU)	64	208	168	247
CARE (GPU A100)	2+26	59	45	247
CARE RF (CPU)	64	225	186	247
CARE RF (GPU A100)	2+20	65	46	247
Musket	64	193	187	138
SGA	64	417	405	37
Karect	64	5055	-	240
Bcool	64	380	376	43
Lighter	64	44	37	16
BFC	64	93	92	108

Tab. 6.4.: Total runtime of correction of dataset A4. Runtime is given in minutes.

The GPU version of CARE is more than three times faster than the CPU version. Lighter and BFC are faster than CARE CPU. Lighter is faster than CARE GPU. The slowest tool is Karect. Its performance bottleneck are accesses to two temporary files used as double buffer of a total size of around 2.8 terabyte. The temporary files

are required because the available system memory is not sufficient for Karect. The available PCIe storage was not large enough to use with Karect.

The benchmarks using the PCIe SSD identify programs whose performance is limited by file accesses. Lighter and CARE benefit the most from the fast storage and achieve speed-ups between 18% and 30%. For other tools the improvements are only few percents. In general, CARE may require file accesses in all phases of its algorithm. The obvious accesses are reading the input file in the construction phase, and writing the output file in the merge phase. During the merge phase, the input file is read a second time to obtain the read headers. They are not used by CARE during error correction and are thus not stored in memory to have more room for hash tables and temporary corrections. Depending on the available system memory, the latter may be spilled to disk which leads to write accesses in the correction phase, and read accesses in the merge phase. This is the case for dataset A4. Karect could not be run on SSD. Because of the vast amount of file accesses, around 12 TB read and 10 TB written, it should greatly benefit from fast storage.

In terms of reported memory usage, CARE has the highest memory consumption. However, in general the measured memory is not strictly required. At its core, CARE needs the reads and the hash tables to be located directly in memory. The remaining free memory, if any, is then utilized to store temporary results. On systems with less memory this leads to results being stored on disk instead at a cost of performance. A similar principle applies to GPU memory. GPU versions of CARE are able to occupy close to the maximum of available GPU memory, but it is not a hard requirement to be able to run the program. CARE attempts to cache as much read data as possible on the GPU for fast access. The remaining read data has to be fetched from slower system memory.

6.7 Multi-GPU Performance

Multi-GPU systems provide greater compute power and an increased amount of available device memory compared to single-GPU systems. In the ideal case of a trivially parallelizeable program without dependencies between the workloads per GPU, the performance scales linearly with the number of used GPUs. Furthermore, an increased total device memory may allow to fit more data into fast GPU memory, avoiding frequent transfers over the slow PCIe bus. Although accesses to memory resident on a remote GPU come with reduced throughput, modern multi-GPU systems typically use interconnects between GPUs that provides higher transfer rates

than accesses over PCIe. System M2 was used for multi-GPU benchmarks with datasets R2 (D.melanogaster, 75.9 million reads), and A4 (Human, 914.7 million reads). While dataset R2 is larger than dataset A1, it is still small enough to fit both reads and hash tables into the memory of a single GPU. This allows to measure the performance using different data distribution schemes with different numbers of GPUs. Dataset A4 is the largest of our datasets and requires the use of multiple GPU to keep both reads and GPU-sided hash tables in device memory. Furthermore, system M2 provides a large amount of system memory and fast file access speed which can increase the performance especially for large datasets like dataset A4.

To submit work to multiple GPUs, there are generally two programming approaches. The first approach is to use a dedicated CPU thread per GPU. In the second approach, a single CPU thread orchestrates the work of multiple GPUs. As we have seen in the previous single-GPU benchmarks, the usage of GPU hash tables achieves the best performance with four threads. However, in a multi-threaded environment the CUDA driver may use locks to coordinate access to internal resources. Implementing the first approach would require to use four threads per GPU, i.e. 32 threads with 8 GPUs, and could lead to frequent locking. Thus, we chose to implement the second approach which uses up to a total of four threads with any number of GPUs.

The development of our multi-GPU implementation uncovered missed opportunities for optimizations. These include sequence encoding and quality encoding on the GPU during read-storage construction, and faster result processing in the merge phase that is part of both the CPU version and the GPU version. Those optimizations have then been incorporated in the latest version of CARE and can yield performance improvements of more than 40%. Unfortunately, at the time of optimization, system M1 was no longer available for benchmarks. To better evaluate the made improvements, we will not only present multi-GPU benchmarks of dataset A4, but also compare single-GPU runs and CPU runs with improvements and without improvements on system M2.

6.7.1 Dataset R2

We begin with the performance evaluation of dataset R2. GPU hash tables and a batchsize of 8192 per GPU were used. The reads require 9.3 GB. The size of hash tables is 12.8 GB. Random Forests were disabled.

Figure 6.12 shows how data distribution affects the multi-GPU speedup for dataset R2. The single-GPU execution time of the correction phase is $66s$. As expected,

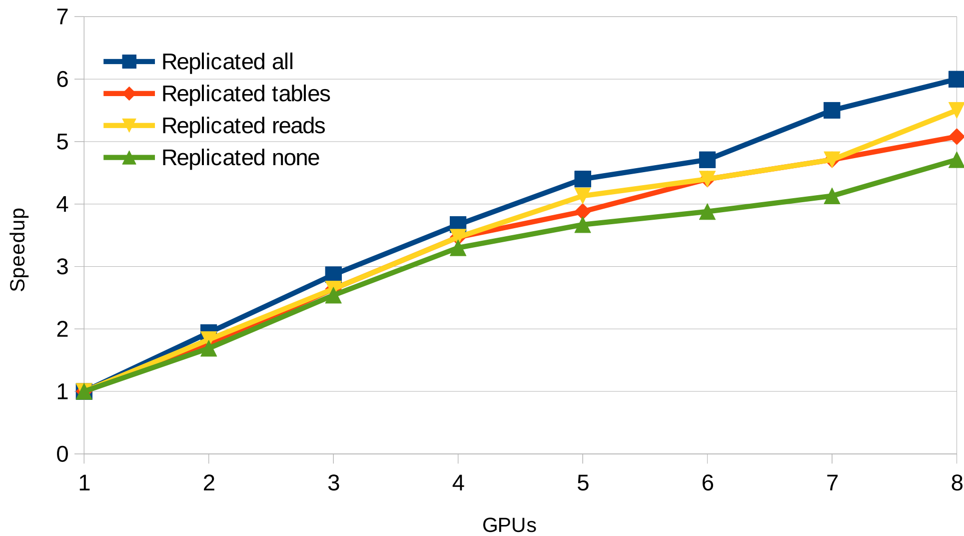


Fig. 6.12.: Multi-GPU speedup of the correction phase of R2 with GPU hash tables for different data distribution schemes. A single thread is used.

avoiding all communication by replicating all data of hash tables and reads on each GPU leads to the best performance. Similarly, no replication causes the greatest amount of communication overhead. Using an even-share distribution of reads (i.e. replicated tables) is slightly worse than distributing the tables.

The efficiency of using multiple GPUs decreases with the number of GPUs. There are a number of factors in play here that limit the performance, namely CPU throughput, inter-GPU communication, GPU load imbalances, and NUMA effects. Those factors are best explained by taking a look at profiler reports generated with the NVIDIA Nsight Systems profiler.

Figures 6.13 and 6.14 show the profiler timeline with 8 GPUs for the hashing and the correction of a batch of 65,536 reads, respectively, with all data replicated on each GPU. A single thread is used. The pictures are rotated to the left by 90 degrees. The top of each picture contains 8 rows to show the activity of the respective GPU (green: host to device transfer, blue: kernel, purple: device to host transfer, white: idle). Below the black bar are section markers. The last row shows the CUDA API calls made by the CPU thread (blue: kernel launch, green: synchronization, red: memory operation).

Typically, CUDA API calls for kernel launches and memory copies are asynchronous and may return before the corresponding GPU operation is complete. Thus, while some kernel is executing on a GPU, the next kernel launch can be submitted on the CPU ahead of time. With n GPUs, we aim to submit kernels to different GPUs

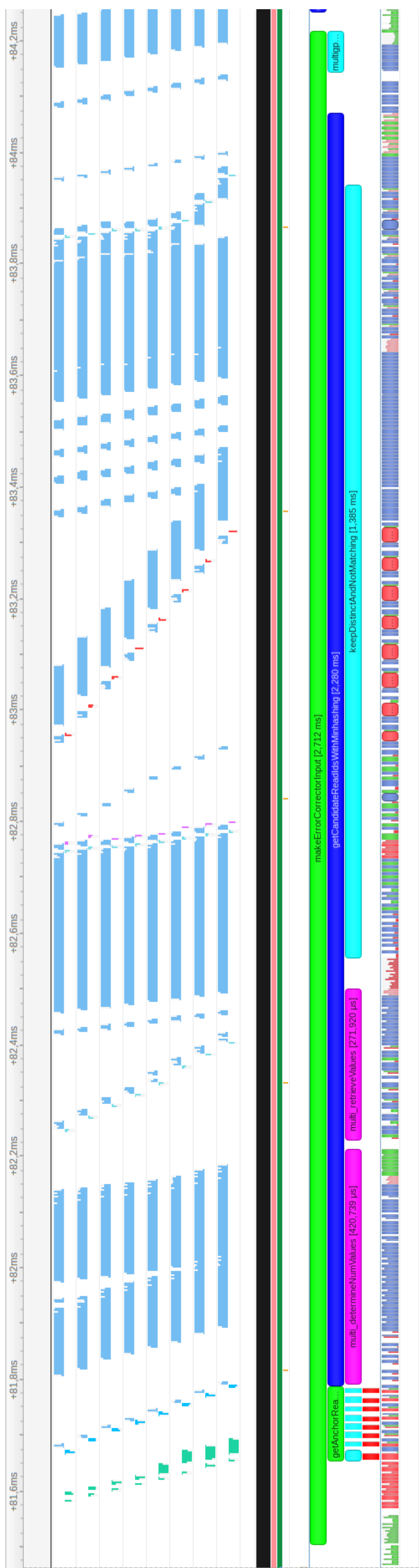


Fig. 6.13.: Timeline to determine the candidate lists per anchor.

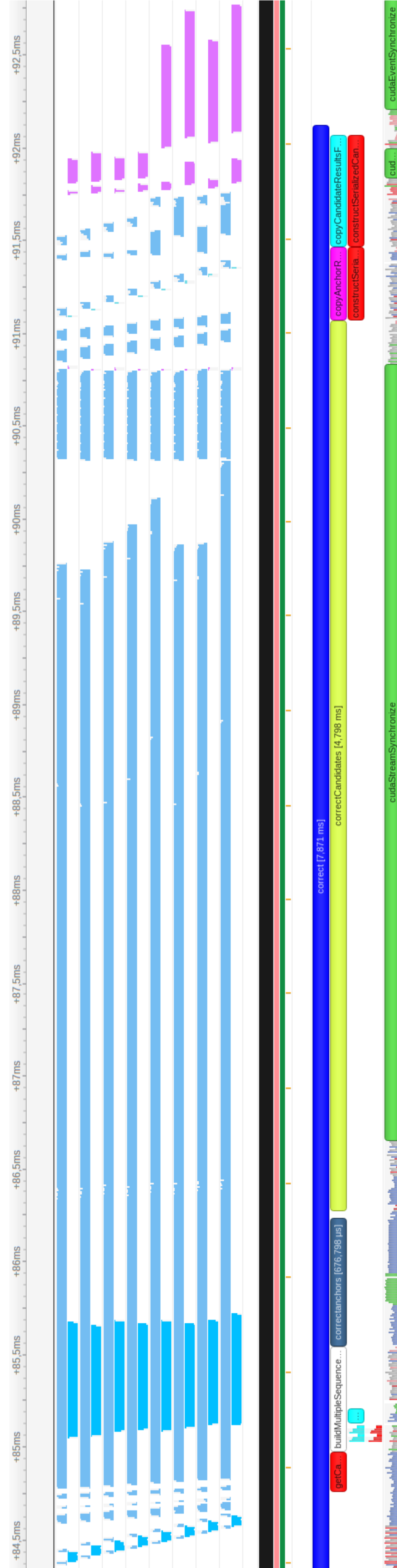


Fig. 6.14.: Timeline to correct the anchors.

in round-robin fashion. Still, each kernel launch takes time. The time between kernel launches to the same GPU effectively increases by a factor of n, and leads to the observable stair-case patterns which indicate delayed kernel execution on the different GPUs in the left-hand picture. This can become a bottleneck for kernels of short duration. For example, a short kernel A on GPU 1 could have finished before the next kernel B can be launched on GPU 1, because the CPU may still be submitting kernel A to other GPUs. Ultimately, this leads to GPU idle time. In our case, the launch bottleneck affects many computations for anchor reads (8,192 per GPU), for example when computing a prefix sum of the number of candidates per anchor to obtain the corresponding segment offsets. The ratio between launch overhead and kernel runtime could be improved by using larger batches. In general, however, the batch size cannot be made arbitrarily large as this increases the memory footprint. CUDA graphs could be used to mitigate kernel launch overheads to some degree. CUDA graphs allow to define a workflow graph with fixed kernel parameters whose execution is possible with reduced overheads. However, in our case many kernels depend on the total number of candidates per batch which may be different for each batch. To incorporate these kernels into a CUDA graph, one would need to frequently update the graph or refactor the kernels. We did not attempt to use CUDA graphs. The usage of up to four threads which process GPU batches helps to reduce GPU idle time caused be scheduling overheads.

Load imbalances can occur during correction of a batch because the number of candidates per anchor can be different which leads to different processing times per GPU for alignments and MSA operations. It can be observed in the right-hand picture after the longest stretch of blue GPU activity. NUMA effects caused by the system topology can be seen at the end of the right timeline. When transferring the corrected sequences back to host memory (purple GPU activity), the transfer of roughly the same amount of data takes significantly more time from a set of four GPUs compared to the other set of four GPUs.

Next, we focus on the communication overhead. Figure 6.15 shows the profiler timeline of the code section that determines the number of hash table results per anchor with even-share tables. Both sequences and sequence lengths need to be broadcast from each GPU to each other GPU. Then, the hash table kernel is executed, and result sizes are transferred back to the respective GPUs where the per-GPU counts are summed up. Each transfer is an all-to-all communication involving n^2 copy operations. Since the transfer sizes are small, the communication steps are bottle-necked by CPU-side API throughput. Overall, the additional steps required for multi-GPU execution take up around 80% of the execution time for this piece of the algorithm.

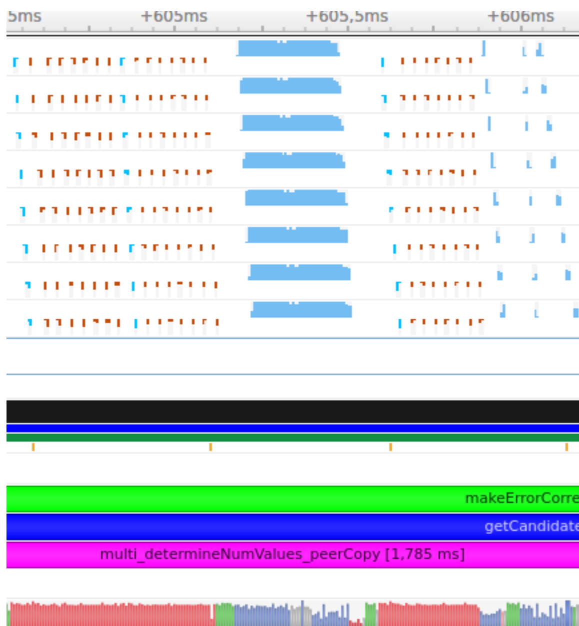


Fig. 6.15.: Multi-GPU hash table access with all-to-all copies using CUDA API calls.

Fig. 6.16.: Multi-GPU hash table access with all-to-all copies using copy kernels.

The communication costs can be reduced by using custom copy-kernels. Within a kernel, it is possible to directly access the memory of other GPUs. Since the eight GPUs are fully-connected, all remote transfers will use NVLINK connections instead of the PCIe bus. Thus, the n^2 copy API calls per all-to-all can be replaced by n copy-kernels which copy the data of one GPU to all the other GPUs. This increases the CPU throughput and overall performance and is the approach we used for all our multi-GPU benchmarks. Figure 6.16 shows the corresponding timeline with copy-kernels.

6.7.2 Dataset A4

First of all, we conducted CPU benchmarks and single-GPU benchmarks with our previously mentioned latest optimizations enabled and disabled. The host memory limit was set to 512 GB. The single-GPU pipeline uses CPU hash tables, 20 producer threads and 2 consumer threads. The batchsize was set to 8192. 2-bit quality scores were used. Those benchmarks serve multiple purposes. Besides the quantification of improvements, it shows the hardware capabilities of system M2. The increased amount of system memory avoids spilling results to disk and allows to construct all 48 hash tables with two passes over the reads. The faster file system accesses

improve reading the input file and writing the output file. And last, the benchmarks give a performance baseline to evaluate the multi-GPU approach.

Table 6.5 shows the measured program execution times. The GPU version greatly benefits from the improved merge phase. It is up to 42% faster than the previous version. The CPU version is limited by correction performance which diminishes the relative performance improvements of the revised output construction.

	Threads	Time (Old)	Time (Improved)
CARE (CPU)	64	179:29	169:41
CARE (single-GPU)	2+20	32:02	18:32
CARE RF (CPU)	64	198:51	189:03
CARE RF (single-GPU)	2+20	33:59	19:44

Tab. 6.5.: Comparison of the improved versions on M2. The single-GPU version uses CPU hash tables. Times are given in minutes:seconds.

Next, we shift the focus to multi-GPU execution. First, Table 6.6 presents the the runtime of multi-GPU distributed hash table construction.

GPUs	1	2	3	4	5	6	7	8
Time	-	-	82	75	56	29	26	24

Tab. 6.6.: Runtime in seconds for GPU minhasher construction with multiple GPUs. The construction of CPU hash tables with a single GPU takes 220 seconds.

The hash tables take up 163 GB. Thus, at least three GPUs are required to store the tables in device memory. Recall that CARE uses a two-phase hash table construction where over-occupied buckets are removed in the second phase. With 5 or less GPUs, there is not enough GPU memory available to use as temporary storage. This requires the use of managed memory which reduces performance compared to the execution times with 6 or more GPUs. In addition, the performance gain for construction with any number of GPUs is limited. For the second phase one host thread per GPU is used because warpcore performs blocking operations internally that would prevent issuing work to multiple GPUs when only one thread were to be used for all GPUs. However, the deallocation of the initially constructed multi-value hash tables acts as a global execution barrier and blocks the calling thread until all work on all GPUs is completed. This causes frequent execution stalls.

Next, we present benchmarks for full program execution. The previous benchmarks for dataset R2 suggest that one should minimize inter-GPU communication. With the memory requirements of GPU hash tables of dataset A4, distribution of hash

tables amongst the GPUs is inevitable. However, with enough GPUs the per-GPU portion of hash tables is small enough to be able to replicate the reads on each GPU. Using 8 GPUs, the hash tables consume around 21 GB per GPU. The reads with 2-bit quality scores require a total of 47 GB. This leaves enough memory to perform the correction. With four threads, 8 GPUs, and disabled Random Forests, the time spent in the correction phase is 69s with replicated reads and 90s with distributed reads.

To conclude the performance evaluation of CARE, we determined the fastest configurations for dataset A4 depending on the number of GPUs and present the individual contributions to runtime in Table 6.7.

<table>
<tr><td>GPUs</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Config</td><td>A</td><td>B</td><td>B</td><td>C</td><td>C</td><td>C</td><td>C</td><td>D</td><td>D</td></tr>
<tr><td>Load input file</td><td>03:55</td><td colspan="8">02:10</td></tr>
<tr><td>Table construction</td><td>18:35</td><td colspan="2">03:40</td><td>01:22</td><td>01:15</td><td>00:56</td><td>00:29</td><td>00:26</td><td>00:24</td></tr>
<tr><td>Correction (no RF)</td><td>141:29</td><td>08:37</td><td>07:40</td><td>02:45</td><td>02:10</td><td>01:56</td><td>01:41</td><td>01:15</td><td>01:09</td></tr>
<tr><td>Correction (RF)</td><td>160:51</td><td>09:48</td><td>07:48</td><td>03:54</td><td>02:50</td><td>02:28</td><td>02:10</td><td>01:41</td><td>01:31</td></tr>
<tr><td>Sort results</td><td>02:42</td><td>01:05</td><td colspan="7">00:15</td></tr>
<tr><td>Merge + Output</td><td colspan="9">3:00</td></tr>
<tr><td>Total (no RF)</td><td>169:41</td><td>18:32</td><td>16:45</td><td>09:32</td><td>08:50</td><td>08:17</td><td>07:35</td><td>07:06</td><td>06:58</td></tr>
<tr><td>Total (RF)</td><td>189:03</td><td>19:43</td><td>16:53</td><td>10:41</td><td>09:30</td><td>08:49</td><td>08:04</td><td>07:32</td><td>07:20</td></tr>
<tr><td>Speedup (no RF)</td><td>1.0</td><td>9.2</td><td>10.1</td><td>17.8</td><td>19.2</td><td>20.5</td><td>22.4</td><td>23.9</td><td>24.4</td></tr>
<tr><td>Speedup (RF)</td><td>1.0</td><td>9.6</td><td>11.2</td><td>17.7</td><td>19.9</td><td>21.4</td><td>23.4</td><td>25.1</td><td>25.8</td></tr>
</table>

Tab. 6.7.: Best configuration depending on the number of GPUs. (A) CPU version – 64 threads, (B) GPU version with CPU hash tables and replicated reads – 2 consumers and 20 producers per GPU, (C) GPU version with GPU hash tables and distributed reads – 4 threads, (D) GPU version with GPU hash tables and replicated reads – 4 threads. Runtimes are given in [minutes:seconds].

In general, the parts of CARE that are amenable to GPU parallelization, table construction and read correction, achieve impressive speedups compared to a purely CPU-based implementation with 64 threads. Using a single GPU with CPU-sided hash tables already gives a great speedup of more than 16 in the correction phase. Yet, this approach should be avoided with multiple GPUs, if possible. It requires to scale the number of producer threads with the number of GPUs to deliver enough input to the GPUs. With that many threads, runtimes may become unpredictable and performance becomes hard to analyze. For example, correction with Random Forests on two GPUs takes almost the same time than without Random Forests whereas with a single GPU the performance using the Random Forests is 12% slower. Still, given the memory constraints we have no other options to make use of a second GPU.

With three to six GPUs, the best performance is achieved with GPU hash tables and distributed read data with four threads. Replicating the reads either led to out-of-memory errors for any number of threads (3-5 GPUs), or required the use of only one thread instead of four threads (6 GPUs; the replicated version with a single thread took 02:16 minutes in the correction phase without RF). With seven or eight GPUs CARE is able to use four threads in both the replicated case and the distributed case. As explained previously, replicating the reads provides better performance.

To sort the results of dataset A4 using a GPU, more than 100 GB of device-accessible memory are required. In the single-GPU case, managed memory is used to overcome the memory limitations. When the combined memory of multiple GPUs is able to provide the required memory, we use the CUDA virtual memory management API to create a contiguous allocation of sufficient size that is backed by the physical memory of multiple GPUs. Compared to the single-GPU case, this uses NVLink inter-GPU accesses instead of PCIe communication with the host and in general avoids the managed memory page migration overhead which leads to improved performance.

Considering the total program execution time, the multi-GPU efficiency decreases with the number of GPUs since the performance bottleneck is shifted to different parts of the program, namely file input and result construction. The times are negligible for the CPU version, but already with just one GPU, the GPU-parallelized code sections account for only around 50% of the total runtime. With 8 GPUs, this decreases to 20%. The achieved multi-GPU efficiency is 33%. For further improvements, we would need to focus on the input step and merge step. For example, both steps perform sequence file parsing which could also be accelerated with GPUs. However, the actual file accesses can only be improved by using faster hardware.

6.8 Proof-of-concept: Sequence parsing on the GPU

To extract usable sequencing reads from a decompressed input file, two steps are required. First, a chunk of raw data must be fetched from the file. Then, this data must be parsed according to the file format. In the case of FASTA and FASTQ files, parsing involves identifying line ends and the type of data in this line (header, sequence, or quality), and copying and concatenating the lines into contiguous output buffers which store headers, sequences, and quality scores, respectively. Our sequence file reader parses the data sequentially on the CPU. This may be sufficient

if file accesses are slow, for example when reading from an ordinary HDD with transfer rates of about 100 MB/s. However, during our experiments on machine M2 with fast file accesses, we noticed that the actual parsing of the raw data can take up to 50% of the read extraction time. If the read data is intended to be used on the GPU, it may be feasible to perform the parsing step directly on the GPU to benefit from its high memory bandwidth and parallel processing capabilities. Depending on the actual performance values, GPU parsing could even be beneficial if the results should be stored in host memory. In this section, we will present a proof-of-concept of this approach.

For simplicity, our experiment only focuses on the FASTQ format which allows an easy identification of the read number and the type of line data in a parallel environment based on the line number (1. header, 2. sequence, 3. separator, 4. quality scores). In general, sequences in FASTA files can span multiple lines and different numbers of lines, which makes it more difficult to parse in parallel. In addition, we do not check for inconsistencies in the data, for example empty lines or mismatching lengths between a sequence and its quality scores. Furthermore, we only consider the parsing of a single chunk of data which fully contains all the reads. A real parser would need additional book-keeping to handle the cases where reads are split between multiple chunks of data.

The actual device-side algorithm can be composed of a few standard parallel algorithms like reduction, stream-compaction, and prefix-scan, all of which are provided by the CUB library. We use a parallel reduction to count the number of new-lines ('\n') in order to allocate an exact amount of memory for the following steps without over-provisioning. Next, we use stream compaction to find the actual positions of new-line characters. Subsequently, the line type and the number of data bytes per line are determined. What is left to do is to copy the line segments into the correct output buffers according to their line type. An inclusive prefix-sum over the segment sizes is performed per line type to compute the destination offsets in the respective target buffer as well as the total size of the target buffer. After resizing the buffers accordingly, the line segments are copied to the respective positions.

For our benchmark, we used machine M2 with a single GPU and a FASTQ file comprising of the first million of reads of dataset A4. The corresponding file size is around 219 MB. The aim is to produce five buffers. Three buffers store flat arrays of headers, sequences, and quality scores. The two remaining buffers store the per-read begin offsets of headers and sequences, respectively. As CPU reference, we implemented a minimal sequential CPU version for FASTQ parsing.

We measured a file reading time of 60 ms which corresponds to a transfer rate of around 3.6 GB/s. The minimal parsing of that data on the CPU took 52.36 ms. The duration of the parsing step of CARE's file reader, which performs multi-chunk parsing and consistency checks was around 62 ms. To be able to parse the data on the GPU, it needs to be present in device memory. The involved data transfer took 8.96 ms. The subsequent parsing step required 7.82 ms. A final transfer of the parsed data back to the host took 9.14 ms. In total, using the GPU to produce results on the host required 25.92 ms which is twice as fast as parsing on the CPU. If the parsed data can be left on the device, the last transfer can be omitted, resulting in an execution time of 16.78 ms. This is around 3.6 times faster than parsing the raw data on the CPU and transferring the parsed data to the GPU. Please note that we did not include device memory allocation times of 3.5 ms, as one would re-use the same buffers for multiple chunks in practice.

Table 6.8 gives an overview of the expected speedup with GPU parsing for the case of host-sided results, depending on different file transfer rates. Table 6.9 shows the corresponding speedup for device-sided results. For machines with slow file storage the benefits of GPU parsing are negligible. Yet, if fast file accesses are possible, parsing the raw data on the GPU can give a speedup of more than 33%. In the context of CARE, a GPU-accelerated file parser could potentially reach greater speedups. This is because the parsed data which is transferred back to the host would be of lesser size since at least the sequences would be 2-bit-encoded.

Transfer rate [MB/s]	128	256	512	1024	2048	4096
File access [s]	1.7109	0.8555	0.4277	0.2139	0.1069	0.0535
CPU parsing [s]	0.0524					
H2D+GPU parsing+D2H [s]	0.0259					
Total GPU speedup	1.0153	1.0301	1.0584	1.1105	1.1995	1.3338

Tab. 6.8.: Speedup of GPU-accelerated FASTQ parsing when the parsed data should be stored in host memory.

Transfer rate [MB/s]	128	256	512	1024	2048	4096
File access [s]	1.7109	0.8555	0.4277	0.2139	0.1069	0.0535
CPU parsing + H2D [s]	0.0615					
H2D + GPU parsing [s]	0.0168					
Total GPU speedup	1.0259	1.0512	1.1006	1.1938	1.3614	1.6358

Tab. 6.9.: Speedup of GPU-accelerated FASTQ parsing when the parsed data should be stored in device memory.

It is worth pointing out that in case of multi-chunk processing, the overall execution time of both approaches could be further reduced by using separate threads for

file accesses and parsing. This would allow overlapping parsing the current chunk with the loading of the next chunk. Additionally, using a GPU for parsing enables more potential improvements. For the parsing of compressed files less data needs to be read from file. However, its sequential decompression on the CPU is slow, as is shown in Table 6.1. If a compression algorithm were used which allows for parallel decompression, the decompression could take place on the GPU. This would lead to reduced file access times, reduced transfer times between host and device, and faster decompression. And last, NVIDIA GPUDirect Storage could be used on compatible systems to directly load file data into device memory using a direct memory access (DMA) path which does not require staging the data in host memory.

7 Conclusion

NGS datasets are affected by errors. While there exists a variety of tools which tackle the problem of error correction in preparation of downstream analysis, its results can be negatively impacted by the presence of false-positive corrections. CARE is an MSA-based tool for error correction of Illumina datasets which can utilize pre-trained random forests for highly accurate sequence modifications.

The evaluation shows that CARE produces around two orders-of-magnitude fewer false positives than state-of-the-art error correction software. At the same time, the number of true positive corrections is on par with those of other tools. This is beneficial for k-mer analysis and de-novo genome assembly. The CPU version of CARE has been integrated into the quality control software RabbitQCPlus 2.0[82]

The processing of CARE is performed in highly parallel fashion to exploit the capabilities of modern hardware. The usage of GPUs allows for faster, more efficient computations. However, achieving peak performance requires both a fast CPU and GPU, as well as sufficient device memory. With low CPU performance, it may be difficult to keep a fast GPU utilized. GPUs with small device memory require more frequent data accesses over slow interconnect. Depending on the used hardware different parallelization strategies may be required. For best performance, all data should reside in GPU memory, and inter-GPU communication should be kept at a minimum. On a modern multi-GPU server, the Random Forest-based correction of a Human dataset with 30x coverage takes 3 hours with the CPU version of CARE, 20 minutes with a single GPU, and 7.5 minutes with 8 GPUs. With decreasing runtimes for compute intensive portions of the program, the performance of other parts like file IO and sequence parsing becomes increasingly important. Performance evaluation shows a speedup of up to 40% when CARE operates on a fast PCIe SSD instead of an ordinary hard disk drive.

Further research could be conducted in two areas. On one hand, there exist other sequencing technologies such as Oxford Nanopore or PacBio which are long-read platforms. Tackling the error correction of those kind of reads would introduce new challenges due to significantly longer reads, different types of sequencing errors like insertions and deletions, and higher error-rates. For instance, the current alignment computation via shifted hamming distance would no longer be viable. It would

need to be replaced by an actual semi-global alignment. On the other hand, while CARE is able to use random forests for improved correction quality, other machine learning techniques may prove beneficial, as well. For example, our MSAs could be interpreted as a multi-channel image which can be passed to a deep neural network for the detection of sequencing errors. Our hand-selected features which are currently used for the random forest may not be optimal whereas a neural network could learn different, better features from the MSAs.

Part II

Read extension

8 CAREx: Context-aware read extension

Short-read sequencing platforms such as Illumina produce reads of a few hundred basepairs. Some bioinformatic applications, however, work better with longer reads. For example, a common problem of genome assembly is repeat resolution. Repeats can only be resolved accurately by reads which are longer than the repeated structure.

Short paired-end reads are often produced with a high depth of coverage. The average distance between the far ends of reads, the so called insert size, can be estimated from the sequencing technology. Given such a read pair, the redundancy of data can be used to reconstruct the nucleotides in-between the reads of the pair. This results in a longer read, a so called pseudo-long read. In the most simple case, the insert size is less than the sum of read lengths of the pair. This indicates that both reads must overlap. Finding pseudo-long reads is more challenging if the insert size is greater than the sum of read length. There are unknown nucleotides which need to be determined in order to produce the pseudo-long read.

Our research focuses on the latter case of non-overlapping reads for NGS sequences from the Illumina platform. CAREx is our newly developed application which produces pseudo-long reads given an input file with short paired-end reads. Its algorithm is based on local assembly with the goal of producing a contig that is delimited by the reads of a read pair. Candidate sequences that could be used for assembly are determined using hash tables. The algorithm of CAREx builds upon the CARE algorithm. It utilizes the same data structures like read-storage and minhasher, as well as the same algorithmic building blocks such as the computation of a shifted hamming distance and the construction of a multiple-sequence alignment. In contrast to CARE, CAREx uses an iterative algorithm which performs multiple passes of hashing, alignment, and MSA construction to produce a pseudo-long read for a single input read pair.

8.1 Related Work

Early work addressed only the simple case, where the insert size is less than the sum of read lengths of the pair. In this case both reads overlap. A pseudo-long read can then be obtained by finding the best overlap with respect to the insert size. This approach is used by FLASH [51], COPE [43], and PEAR [83].

Extending short read pairs to pseudo-long reads is more challenging if the insert size exceeds the sum of read lengths. In this case, there are unknown nucleotides which need to be determined from other overlapping reads in order to produce a pseudo-long read. This problem can be formulated as a small local assembly operation, creating a small contig that is delimited by the two reads of a pair.

Existing approaches which extend a given read until its partner (mate) is found are based on detecting overlaps between k-mers. They can be distinguished by their utilized data structures and specific extension rules. GapFiller [55] and Eloper [73] employ simple seed-and-extend strategies based exact matches using hash tables. Konnector2 [79] constructs a De Bruijn graph from the k-mers of all reads. Subsequently, the De Bruijn graph is traversed for each read pair to find a connecting path, which is then translated into a pseudo-long read. MaSuRCA [84] is a genome assembler that is build around the construction of super-reads. It extends a read on each end base per base as long as there is only exactly one possible base to append. The k-mer spectrum of the input read dataset is used to verify that only one possibility exists. Aside from contig assembly, the construction of pseudo-long sequences can also be used by genome assemblers to bridge gaps between contigs within a scaffold [28, 7, 50]. PLR-GEN [74] creates pseudo–long reads from clustered metagenomic short reads based on given reference genome sequences.

8.2 Algorithm

The computation of pseudo-long reads in CAREx revolves around the processing of so called extension tasks. Given a starting sequence S, an extension task will iteratively append suitable nucleotides to the $3'$ end of the sequence until a stop condition is met. This is performed by constructing MSAs of reads similar to S. Then, a substring of the MSA consensus is used to elongate S.

Let $S1$,$S2$ be two sequences that form a read pair. Let $RC1$, $RC2$ be the reverse complement sequences of $S1$ and $S2$, respectively.

For this read pair, four extension tasks, T1, T2, T3, and T4, are created with different starting sequences, namely S1, RC2, S2, and RC1, respectively. Figure 8.1 shows the initial sequence layout of the four extension tasks, and how those tasks move in $3'$ direction in one iteration of the algorithm.

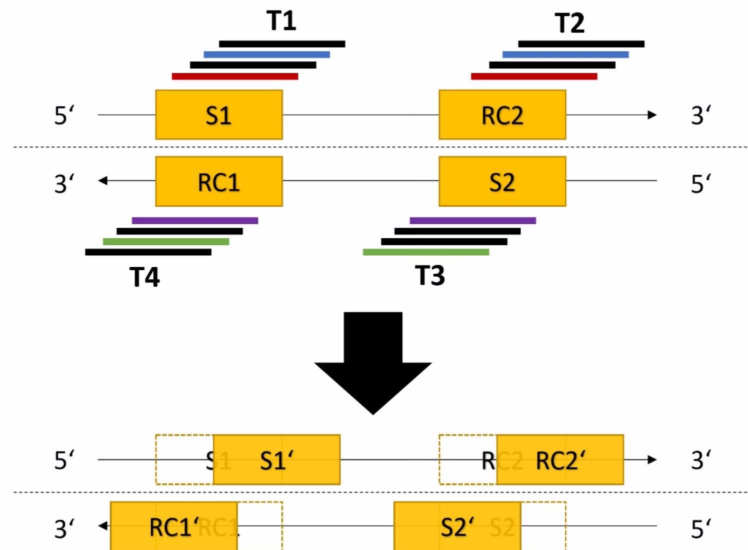


Fig. 8.1.: Layout of the four sequences for extension tasks.

Tasks 1 and 2 are considered partner tasks, as well as tasks 3 and 4. Tasks 1 and 3 are primary task with the goal of finding the end of the pseudo-long read, i.e. task $1(3)$ will stop if $RC2(RC1)$ has been reached. Tasks 2 and 4 are auxiliary tasks connected to tasks 1 and 3, respectively. They terminate as soon as their partner task stops. In addition, each of the four tasks can end if it is no longer possible to perform an extension of the sequence. The tasks are processed simultaneously. After all four tasks are completed, the pseudo-long read of the read pair is constructed, if possible, from the generated extended sequences of tasks 1 and 3. Furthermore, the generated sequences of T2 and T4 can be appended to the respective ends of the constructed pseudo-long read to extend the read pair in outward direction. Outward extension is not the main focus of CAREx and is disabled by default.

8.2.1 Construction phase

In the construction phase, the dataset is loaded into memory. Hash tables are constructed from the reads to quickly find similar reads to a given query read. The construction phase is identical to the construction phase of CARE, see Section 4.3.1.

8.2.2 Extension phase

This subsection will explain the steps necessary to construct a pseudo-long read for a single read pair after the four extension tasks have been created. The steps are repeated for each read pair in the dataset.

Extension task processing

At the beginning of a processing iteration of a task, its sequence S is queried against the hash tables to retrieve a set C of potentially similar candidate sequences with the intention of constructing an MSA centered around S. In general, this is performed using the same steps as in CARE, namely shifted hamming distance computation, alignment filtering, and MSA construction with refinement, potentially using a read's quality scores. However, there are some differences to CARE in the alignment computation and the alignment filtering, which will be described below.

For the alignment computation using shifted hamming distance, only alignments with positive shifts are considered. This is because the task aims to extend its sequence only on the right end, not on the left end. In addition, valid alignments must overlap with the anchor by at least 50% (default). Within this overlap, at most 5% (default) mismatches are allowed.

Recall that the four tasks of a read pair are processed simultaneously and that auxiliary tasks may have terminated in a previous iteration because extension could no longer be performed. In case both the primary task and its auxiliary task are still running, candidate pairs are identified as in CARE via their corresponding read ids. In addition, candidate pairs are also considered if one of the reads was already used by the partner task in a previous iteration. For example, let $C_1 = \{r_0, r_5, r_8\}$ and $C_2 = \{r_2, r_9\}$ be the candidate sets of task 1 and task 2, respectively. Additionally, let $U_2 = \{r_4, r_{13}\}$ be the set of candidates used in a previous iteration. Then candidates r_5 and r_8 of task 1 are considered paired candidates, as well as candidate r_9 of task 2. Paired candidates are assumed to originate from the same genomic region as the pseudo-long read to be constructed. If either the primary task or its partner task is already finished, all candidates are treated as unpaired. When the alignment filter is applied, paired candidates are kept unconditionally. Unpaired candidates are removed depending on their alignment overlap. Let O_i be the relative overlap size between unpaired candidate i and the anchor S. Let $T = \max_i(\lfloor O_i \cdot 10 \rfloor / 10)$. Then unpaired candidates with $O_i < T$ are removed.

After candidates have been filtered, and a refined MSA M has been constructed in the same manner as CARE, the consensus string of M is used to compute an extended sequence S' from S. Let column i of the MSA be the first column of M that is not covered by the anchor read S. The algorithm determines column $j \geq i$ where $j - i < stepsize$, the coverage of column $j \geq m$, and j maximal. Let $n = j + 1 - i$. Then the extended sequence S' is computed as $S' = S[n : |S|] + consensus[i : i + n]$. That is, S' is obtained by appending a consensus substring of length n to S and removing the first n nucleotides from S. $stepsize$ and m are parameters of the algorithm with default values of $stepsize = 20$, and $m = 3$. In contrast to CARE's usage of the MSA consensus, no elaborate decisions about whether or not to use the consensus nucleotide can be made in CAREx, because there is no fallback to the anchor nucleotide.

There are three different outcomes of the computation of S'.

1. The algorithm may fail to compute S' if either column i or column j do not exist. This can be the case if the number of columns in M is $|S|$, or if all columns to the right of column i have low coverage. If computation fails, the task terminates.
2. When computing S' in a primary task the algorithm may find that the target sequence, i.e. the read's mate, has been reached. In that case, the task terminates, as well.
3. Extension succeeds without reaching the mate. The task's sequence S is updated to S' and the next task iteration begins.

Finalization

Eventually, all four tasks of a read pair will be finished. In a last step, the pseudo-long reads of the tasks are merged to find the pseudo-long read of the read pair, i.e a pseudo-long read that connects $S1$ and $S2$, on any strand. The construction of the final extended read from the four tasks can be controlled by a parameter called *strict mode*. We currently provide three different levels of strictness.

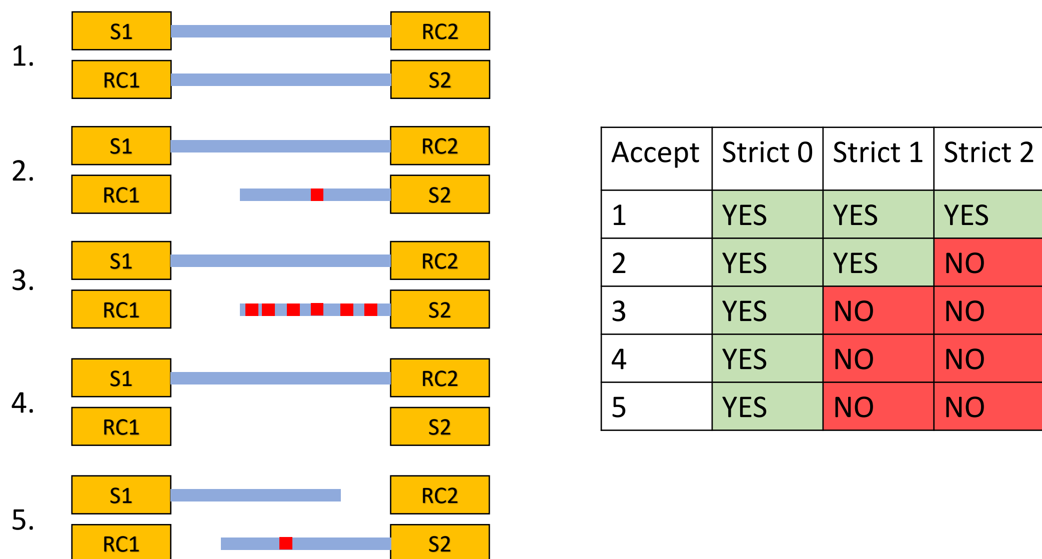
Strict mode 2 is the most restrictive. If both T1 and T3 have finished after finding the target sequence, and both tasks produced a connection of same length between the reads, this connection is used only if the hamming distance between the connections of the two tasks is less than some parameter x. By default, we require an exact match. In all other cases, the read pair remains unconnected.

Strict mode 1 includes mode 2. In addition, mode 1 can handle the case where only one of T1 or T3 found the mate. For example, assume T1 found its mate and the size of the filled gap is s. Then the connection is used if the overlap between the filled gap of $T1$ and the incomplete filled gap of T3 is at least of size y (default: 50% of s), and the overlap contains at least $z\%$ matches (default: 95% of s).

Strict mode 0 is the least restrictive. If either T1 or T3 have finished after finding the target sequence, their corresponding result sequence is used as the pseudo-long read. For some read pairs, neither of both primary tasks may have finished successfully. Yet, they may have computed sufficiently long, but incomplete, pseudo-long reads. If agreeable with the estimated insert size, the result sequences of both tasks are merged to form the pseudo-long read of that read pair. Otherwise, the read pair is not connected. The merge will only be performed if the suffix of one sequence can be overlapped with the prefix of the reverse complement of the second sequence by at least 40 positions with at most 5% mismatches. In case of multiple possible overlaps, the longest overlap is chosen.

Figure 8.2 gives an overview of possible scenarios for different levels of strictness. In any case, if it was possible find a pseudo-long read that connects $S1$ and $S2$, results of T2 and T4 may be used to further elongate it on both ends. This is achieved by simply appending the extended nucleotides of those tasks to the appropriate ends of the pseudo-long read, respecting the correct strand.

In all cases, for positions in the pseudo-long reads which correspond to the original reads the original nucleotides are used, i.e. no modifications to the original reads are performed.



Accept	Strict 0	Strict 1	Strict 2
1	YES	YES	YES
2	YES	YES	NO
3	YES	NO	NO
4	YES	NO	NO
5	YES	NO	NO

Fig. 8.2.: Five different outcomes of extension task processing. Mismatches between the two strands (i.e. non-complementary bases) are marked in red. The right-hand table indicates whether a pseudo-long read can be constructed for a specific strictness.

8.3 Implementation

CAREx is available on Linux as a multi-threaded CPU version written in C++ as well as a GPU version written in CUDA/C++ which can utilize CUDA-capable GPUs for high performance. The overall algorithm is embarrassingly parallel since read pairs can be processed independently. Thus, the main processing loop which iterates over all read pairs is trivially parallelizeable. In the CPU version, C++ threads are utilized for parallelization of the main loop. Each thread produces a list of read ids for which no pseudo-long read could be constructed.

The GPU-accelerated processing of the main loop comes with two challenges in regards of GPU utilization: load imbalances and hash table accesses.

Read pairs are processed in batches. One challenge is the presence of load imbalances in a single batch. Extension tasks may terminate at any iteration which can lead to poor performance when the number of active tasks within a batch is too small to fully occupy the GPU. This problem is solved by using dynamic-sized batches. For each batch there are two sets of tasks, active tasks and finished tasks. The active tasks are processed simultaneously on the GPU. After each extension iteration, terminated tasks are moved from the set of active tasks to the set of finished tasks. Whenever the number of active tasks falls below a predefined threshold, new tasks for yet unprocessed read pairs are added to the set of active tasks until its size reaches the defined batch size.

The finished tasks are handled in similar fashion. Recall that for each read pair there exist four extension tasks. Only when all four tasks are finished, the final pseudo-long read can be constructed. Whenever the number of finished tasks surpasses a threshold, read pairs with four finished tasks are identified, their resulting pseudo-long reads are constructed on the GPU, and the corresponding tasks are destroyed. The thresholds for both sets of tasks are chosen empirically such that small, inefficient GPU workloads involving the two sets of tasks do not occur frequently.

Hash tables may be placed in either GPU memory or CPU memory. The second challenge is to minimize the performance bottleneck of host-sided hash table operations. In CARE, this was achieved by a producer-consumer approach to overlap GPU-sided error correction steps of a batch with CPU-sided hash table accesses of a different batch. In CAREx, multiple processing iterations are performed per batch where hash tables are queried in each iteration. This introduces dependencies between successive iterations and prohibits overlapping hash table operations and read extension operations of the same batch. However, iterations of different, independent batches

may still be processed concurrently, and are indeed processed in that manner in CAREx. This is explained in the following paragraphs.

When hash tables reside in CPU memory, two sets of threads, H and E, are used. Threads in set H perform hash table queries, whereas threads in set E are responsible for read extension. In principle, this approach also follows a producer-consumer pattern. However, there are no dedicated producer threads or consumer threads because of the circular dependency between the two sets as batches are passed between the sets in both directions. Communication between sets is achieved via queues Q_H and Q_E. Batches in Q_H (Q_E) are processed by set H (E).

A thread in set H removes a batch from Q_H, resizes the batch accounting for load imbalance as described above, and performs hash table operations. Last, the batch is submitted to Q_E. When the batch is subsequently processed by a thread of set E one extension iteration is performed. After this iteration, the batch may contain both active tasks and terminated tasks. The latter are moved to the set of finished tasks, which are subsequently processed avoiding load imbalance as described above. The thread finishes the processing of this batch by placing it in Q_H.

Initially, $|E|+|H|$ empty batches are placed in Q_H. The overall processing of batches terminates when the number of processed read pairs equals the total number of read pairs in the dataset. Depending on the actual hardware, one or two threads which perform extension are sufficient, but need to be complemented with 8 or more threads responsible for (slow) hash table lookups.

While this parallelization approach leads to increased performance, memory usage increases as well. This may require a trade-off between processing speed and memory consumption. However, this is only the case for host-sided hash tables. When hash tables are placed on the GPU this elaborate scheme is not required. Instead, similar to CARE, only a few threads are required to keep the GPU utilized, and those threads perform both the hashing step and the extension step.

CAREx has multiple output options. First, outward extensions can be enabled or disabled. Second, the output can consist exclusively of pseudo-long reads. Alternatively, read pairs which could not be extended can be included in the output as well. Lastly, the output can be sorted with respect to the input file. When a pseudo-long read is constructed, it is submitted to a separate worker thread. It then is either immediately written to file in case of unsorted output, or temporarily stored in memory otherwise. The sorting is performed after the extension phase is finished.

9 Evaluation of CAREx

Read extension was evaluated on both simulated datasets and real-world datasets. For both types of datasets, the number of connected read pairs is determined, as well as the accuracy of extensions. In addition, the quality of de-novo assembly using extended real-world reads is evaluated. We compared the results of CAREx to publicly available standalone tools that can produce one pseudo-long read per input read pair. In our evaluation, we did not consider the build-in gap closing functionality of assemblers since they only operate at contig-level. Thus, we chose GapFiller 2.1.2 and Konnector 2.2.4 as competitors for the evaluation.

For simulated datasets the exact location of each read within its reference genome is known. This allows for detailed statistics of the generated pseudo-long reads. For example, the edit distance in the filled gap of the pseudo-long read can be computed. It is also possible to confirm the correct distance between the two reads within their produced pseudo-long read. For real-world datasets, however, no reference positions are known. Yet, some insight can be obtained by aligning the pseudo-long reads to a reference genome. Aside from alignment-based metrics, quality of extended real-world datasets can be measured by investigating the impact on down-stream analysis of extended datasets.

To facilitate an easier evaluation, CAREx outputs the exact read positions within a pseudo-long read. This allows to quickly identify the outward extension and the filled gap. While Konnector2 is also able to perform outward extensions, it is lacking the read positions within the pseudo-long reads which prohibits an accurate evaluation. Thus, no outward extension for Konnector2 is performed. In this case, the reads are located at the begin and at the end of the pseudo-long read. The same is true for GapFiller which does not support outward extension.

The remainder of this chapter is structured as follows. After introducing datasets and program settings used for evaluation, we first present metrics for simulated datasets extended by CAREx. Here, we will show the number of connected reads and their error rate, and analyse the pseudo-long read lengths. Then, the evaluation on real-world datasets is presented. Afterwards, we investigate how read-extension is affected by sequencing errors and by error correction with CARE. The chapter closes with the performance evaluation.

9.1 Datasets

Name	Organism	Cov.	Read pairs	Length	Insert size	stddev
S1	C.elegans	30x	15.0M	100	300	5
S2	C.elegans	30x	15.0M	100	500	10
S3	C.elegans	30x	15.0M	100	1000	30
S4	C.elegans	60x	30.1M	100	1000	30
S5	D.melanogaster	30x	18.0M	100	500	10
S6	Hum. Chr. 14	30x	13.2M	100	500	10
S7	C.elegans	30x	10.1M	150	500	150
R1	D.melanogaster	64x	37.9M	101	598	39
R2	Human Chr 21	33x	6.7M	100	312	14
R3	Human (NA12878)	30x	304.6M	148	546	117

Tab. 9.1.: Simulated (S1-S7) and real-world (R1-R3) datasets used for evaluation.

Simulated datasets were generated using the ART read simulator. Three different reference genomes were used: Drosophila melanogaster (D.melanogaster), Caenorhabditis elegans (C.elegans), and Human (Chromosome 14). We chose datasets with varying properties to investigate the impact on read extension. S1-S6 were simulated using ART's HiSeq 2000 profile which yields reads with a sequencing error-rate of around one percent. Datasets have a coverage of 30x and $60x$, and read length 100. Three sets of insert sizes were used: 300, 500, 1000 with standard deviations 5, 10, 30, respectively. Read lengths of a read pair are included in the specified insert size. That is, the average gap size between the two reads of a read pair is insert size -200. Dataset S7 was simulated with a more recent TruSeq profile and has a larger standard deviation for insert size. Its error-rate is around 0.4%.

Publicly available real-world datasets come from D.melanogaster (SRR988075), Human Chromosome 21 (NA19240 Illumina Data Library[1]), and from a full human genome data (NA12878 / HG001). For the full human dataset, we concatenated the samples from SRR2052337, SRR2052338, SRR2052339, SRR2052342, SRR2052348, SRR2052352, and SRR2052354, to obtain 30-fold coverage. Table 9.1 gives a summary of the used datasets.

[1]https://cloudstor.aarnet.edu.au/plus/s/f0f6cb1385704ae8403dfbf86dd622d8/download?path=%2F&files=Human_NA19240.7z

9.2 Program options

All tools come with a different set of program options and tuning parameters. Usually, there is no setting that works best on all inputs. For GapFiller we specified insert size and standard deviation. Konnector2 was run with a k-mer size of 32. The size of the Bloom filter to represent a De Bruijn graph was set such that the reported false-positive rate is around 0.3%. CAREx used 48 hash tables with k-mer size 20. For both Konnector2 and CAREx we set the maximum allowed pseudo-long read length to $insertsize + 4 \cdot stddev$, and the minimum length to $\max(2 \cdot readlength, insertsize - 4 \cdot stddev)$. Thus, we did not consider the simple case of overlapping reads.

9.3 Evaluation on simulated datasets

The quality of produced pseudo-long reads is presented in terms of number of connected read pairs, number of error-free connections, and error-rate within the filled gap. In addition, we take a brief look at the quality of outward extensions.

Figure 9.1 presents the percentage of connected read pairs for the different strictness levels of CAREx, in comparison to the results produced by Konnector2 and GapFiller. This figure also indicates the number of connected reads whose edit-distance within the filled gap is either 0 or less than 3.

CAREx achieves a high percentage of connected read pairs with up to 99% connections for dataset S7 with strictness 0. As expected, using a more conservative mode (strict mode 1 or 2) decreases the number of successful connections. For strict mode 2, a decrease by around 10% of all pairs can be observed compared to strict mode 0. The majority of connections produced by CAREx have an edit-distance of 0.

In comparison to the competitors, CAREx produces significantly more pseudo-long reads. For all simulated datasets, the total number of connected read pairs by other tools is less than the number of error-free connections produced by our algorithm. Compared to the other tools, GapFiller is not competitive, connecting only a fraction of the pairs. In addition, due to the lack of multi-threading capabilities, we were unable to obtain results for datasets S3, S4, S6, and S7, in reasonable time.

Next, we take a look at the general error-rate of the filled gaps. We define the error-rate as the sum of edit distances in the gap for all connected pairs divided by the sum of expected gap sizes for all connected pairs. Despite the fact that CAREx

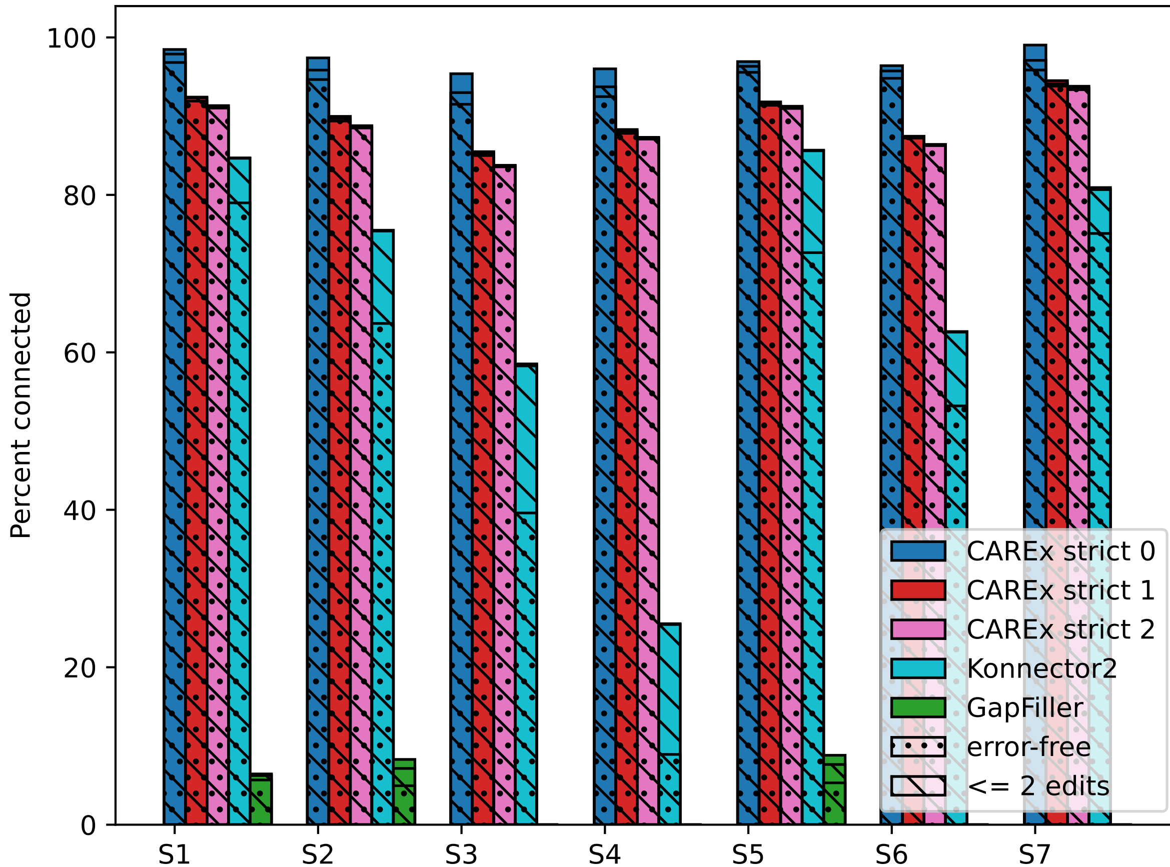


Fig. 9.1.: Percentage of connected reads for simulated datasets. GapFiller did not finish for S3, S4, S6, and S7.

produced the most number of error-free gaps for the simulated datasets, Konnector2 has a smaller error-rate than CAREx with strict mode 0 in four out of seven datasets. On the other hand, error-rates drop significantly when using strict mode 1 and 2 and are better than those of Konnector2. The error-rates for all modes and tools are presented in Table 9.2.

The computation of edit-distance measures the combined effect of two sources of errors in the filled gap. First, an extension algorithm may select the wrong nucleotide to append and thus introduces a substitution error. Second, the algorithm may produce a pseudo-long read of incorrect total length which leads to indels compared to the reference genome region. When conducting the experiments, our assumption was that the reason for inferior error-rates of strict mode 0 compared to Konnector2 is incorrect pseudo-long read lengths. A first indicator is the immediate comparison of error-rates of strict mode 1 and 2 to the error-rates of strict mode 0. The major difference of mode 0 to the other modes is that it does not impose

Dataset	CAREx			Konnector2	GapFiller
	strict 0	strict 1	strict 2		
S1	0.00059	0.00013	0.00006	0.00078	0.00278
S2	0.00081	0.00012	0.00005	0.00063	0.00396
S3	0.00128	0.00009	0.00003	0.00070	DNF
S4	0.00127	0.00010	0.00003	0.00111	DNF
S5	0.00026	0.00006	0.00004	0.00056	0.00371
S6	0.00030	0.00004	0.00002	0.00062	DNF
S7	0.01072	0.00252	0.00151	0.00216	DNF

Tab. 9.2.: Total error rate of filled gaps for simulated reads.

restrictions regarding the length when the extension tasks for both strands produce a result. In contrast, both strict mode 1 and 2 require both strands to produce a result of identical length. Since the number of connected read pairs does only decrease by 10% when using mode 2 instead of mode 0, this means that for the majority of connected pairs extension tasks succeed on both strands.

A second indicator is given by the distribution of produced pseudo-long read lengths. Table 9.3 shows the differences between expected pseudo-long read lengths and observed pseudo-long read lengths for dataset S3. While strict mode 0 produces the greatest amount of connections of correct length, the variation in lengths is greater than for Konnector2. Using strict mode 1 or 2 leads to a more defined peak.

Δ	CAREx			Konnector2
	strict 0	strict 1	strict 2	
< -4	59,937	5,747	1,431	120
-4	4,558	273	66	2
-3	5,336	453	81	3
-2	9,519	554	173	267
-1	26,207	3,356	1,801	2,258
0	14,098,810	12,838,774	12,591,236	8,763,490
1	23,234	3,148	1,541	1,339
2	8,815	583	156	114
3	4,940	282	76	226
4	3,700	275	113	90
> 4	102,653	6,579	2,308	36,719

Tab. 9.3.: Difference between expected gap size and pseudo-long read gap size for dataset S3.

To confirm our assumption regarding the impact of length errors on the overall error-rate, we computed a second set of error-rates that use a modified edit-distance score. The modified edit-distance is computed by subtracting the absolute length difference between expected gap size and produced gap size from the original edit-distance.

Table 9.4 lists the corresponding error rates for the modified edit-distance. In almost all cases, CAREx has better error-rates. In general, the longer the average insert size, the more likely it is for CAREx to fail to find the correct length. In addition, a greater standard deviation, i.e. a large number of possible target lengths, leads to large discrepancies in produced lengths as in dataset S7. This is because CAREx always chooses the first possibility to end extension by adding the mate at the end. Thus, if the length is incorrect, the produced pseudo-long read is more likely too short rather than too long. However, the total number of those cases is small (98% of connections by CAREx on S7 have less than 3 edits). Yet, because of the large insert size deviation, the contributions of those reads to the final error-rate can be significant. Similar observations apply to Konnector2 as well, though to a lesser extend.

Dataset	CAREx			Konnector2	GapFiller
	strict 0	strict 1	strict 2		
S1	0.00035	0.00009	0.00004	0.00071	0.00270
S2	0.00041	0.00006	0.00003	0.00053	0.00385
S3	0.00045	0.00003	0.00001	0.00041	DNF
S4	0.00042	0.00003	0.00001	0.00082	DNF
S5	0.00015	0.00004	0.00002	0.00051	0.00365
S6	0.00020	0.00002	0.00001	0.00050	DNF
S7	0.00029	0.00005	0.00003	0.00032	DNF

Tab. 9.4.: Total error rate of filled gaps excluding length differences.

Last, we present our observations regarding outward extensions of CAREx. As explained previously, Konnector2 does not provide a simple way to extract outward extensions from its generated sequences. Thus, we are not able to compare our observations to the other tools.

Table 9.5 shows the observed minimum length, average length, and maximum length, of produced pseudo-long reads for all datasets with strict mode 0. When averaging the presented values over all seven datasets, the average produced read length is $1.95 \cdot (insertsize + 4 \cdot stddev)$, with a maximum length of $2.67 \cdot (insertsize + 4 \cdot stddev)$. With strict mode 2, we observe an average length and maximum length of $2.03 \cdot (insertsize + 4 \cdot stddev)$, and $2.62 \cdot (insertsize + 4 \cdot stddev)$, respectively.

The error-rate of outward extensions, i.e. to the left of the first read and to the right of the second read, is given in Table 9.6. In direct comparison to the error-rates within the filled gap (Table 9.2), outward extensions show up to two orders-of-magnitude more errors. One possible reason for this could be that there is only a single extension task that produces an outward extension. Thus, there is no

Dataset	Min	Avg	Max
S1	280	591	777
S2	460	1148	1441
S3	880	2509	3195
S4	880	2547	3188
S5	462	1161	1433
S6	461	1111	1433
S7	301	1050	2903

Tab. 9.5.: Pseudo-long read lengths when outward extension is enabled. Strict mode 0 was used.

additional information, i.e. a second task like for filling the gap, that can be used to detect and reject mistakes in extension.

Dataset	Strict 0	Strict 1	Strict 2
S1	0.00243	0.00167	0.00124
S2	0.00325	0.00176	0.00131
S3	0.00499	0.00285	0.00246
S4	0.00455	0.00250	0.00216
S5	0.00112	0.00062	0.00047
S6	0.00128	0.00085	0.00077
S7	0.00377	0.00221	0.00144

Tab. 9.6.: Error-rate of outward extended regions.

9.4 Evaluation on real-world datasets

Two different evaluations are performed with real-world dataset. First, as with simulated reads, the number of connected read pairs and the error-rate in the filled gap is determined. This is achieved by mapping extended reads to a reference genome. Second, applicability of extended reads in de-novo assembly is investigated.

For the real-world datasets the program options were modified. Recall that in CAREx original read positions in the pseudo-long reads are left unmodified. Konnector2, however, performs error correction on these positions by default. These differences might impact the mapping process and the assembly. Thus, for our real-world evaluation we disabled the built-in error correction for Konnector2 with the parameter `-preserve-reads`. The Bloom filter size was set to $15G$, $3G$, and $200G$, for R1, R2, and R3, respectively. This results in a reported false-positive rate of 0.26%, 0.29%, and 0.31%, respectively, which is similar to the rate set for simulated datasets. For dataset R3, the minimum and maximum insert size was set to 300 and 1100,

respectively. To be able to fit the whole dataset in GPU memory, CAREx used 2-bit quality scores for dataset R3.

9.4.1 Edit statistics

In a real-world setting no reference positions are known. To be able to determine the error-rate in the gap, extended reads were aligned to a reference genome using BWA 0.7.17 [40]. Alignments were filtered with SAMtools [12] to remove secondary alignments and supplementary alignments which results in at most one alignment per read. In addition, clipped alignments are not considered for evaluation. In a clipped alignment, parts of the read could not be aligned. Positions of the filled gaps were subsequently extracted from the alignments and compared to the reference genome to compute the edit distance. Note that since only edits within the filled gaps should be considered, the total edit distance reported by BWA cannot be used.

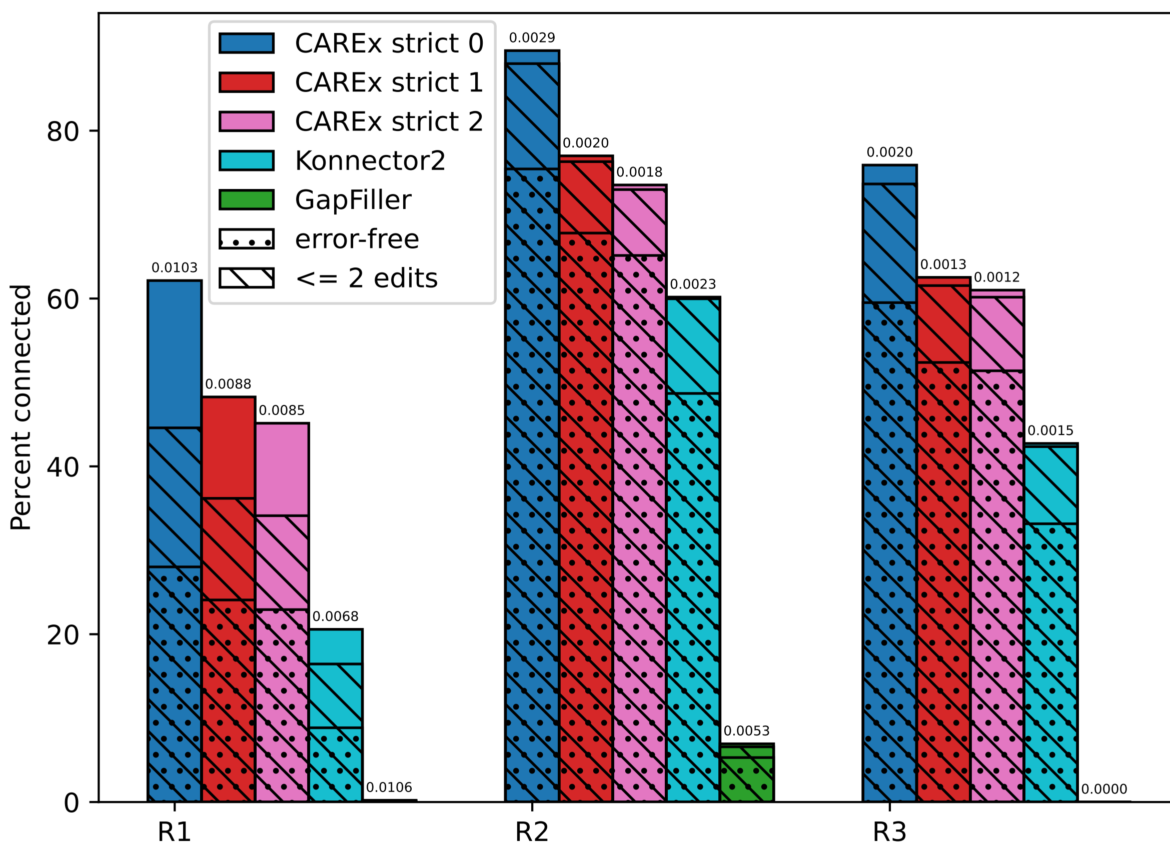


Fig. 9.2.: Percentage of connected read pairs of real-world datasets. Error-rate is given at the top of each bar.

Figure 9.2 shows the percentage of connected read pairs and the corresponding fraction of edit-free filled gaps, as well as the fraction of filled gaps with edit-distance ≤ 2. Additionally, the error-rate is reported, as well. Neither Konnector2 nor CAREx were run with enabled outward extension. CAREx produced the most connected read pairs and the highest amount of error-free connections on all three datasets. In terms of error-rate, results of Konnector2 are better than CAREx with strict mode 0 because the fraction of pseudo-long reads with more than two errors is smaller. This is especially true for dataset R1 where the error-rate of CAREx remains greater than Konnector2's regardless of the selected strictness. Still, more restrictive extension leads to a decrease in error-rate, and reaches better values compared to Konnector2 on R2 and R3. Note that edits may also be caused from genomic variations, for example single nucleotide polymorphisms (SNPs), compared to the used reference genome. Thus, the actual number of wrong nucleotides, for all tools, may be different. As with simulated reads, the results of GapFiller are not competitive. We did not attempt the extension of dataset R3 with GapFiller.

9.4.2 De-novo assembly

De novo assembly was performed with SPAdes v3.13.1 and analyzed with QUAST v5.0.2. For our use-case SPAdes provides three options to specify its input files: single-end reads (parameter `-s`), paired-end reads (parameter `--12`), or merged reads (parameter `--merged`). Finding the best evaluation approach is difficult since there are many possible combinations of settings and used input files. Both Konnector2 and CAREx produce two sets of reads, pseudo-long reads and remaining reads. Should the extended reads be used as merged reads or single-end reads? Should remaining reads be used as single-end reads or paired-end reads? Should they be used at all?

The choice of parameters is further complicated when considering outward extensions. When outward extension is enabled, output files consisting of unextended reads are used differently by CAREx and Konnector2. If a read pair in CAREx could not be extended, both reads are written to the file of unextended reads. This produces a paired-end file of unextended reads. In Konnector2 with enabled outward extension, read pairs that could not be processed are split into two single-end reads that are subsequently extended on both ends individually. Only those single-end reads that could not be elongated in this second step are written to the left-over file. Thus, this produces a single-end file.

The third complication is that there are different metrics that can be utilized to compare assemblies. For example, should one prefer assemblies with less contigs, or with longer contigs, or with lower error-rate?

For this evaluation, two different sets of assemblies were produced without considering outward-extension. In the first set, extended reads were specified as single-end reads, and remaining reads as paired-end reads. In the second set, extended reads were instead specified as merged reads. For assembly metrics, we compared the number of large contigs ($\geq 50,000$ bp), the N50 value, and the number of misassemblies as well as the total error-rate. Only Konnector2 and CAREx were considered. GapFiller was excluded because of the previously shown sub-optimal results. The assembly of R3 was not attempted.

R1				
	CAREx			Konnector2
	strict 0	strict 1	strict 2	
contigs $\geq$ 50,000	667	660	664	659
N50	99,411	105,136	94,000	110,321
misassembled contigs	837	764	769	712
misassembled contigs length	54,708,668	56,368,222	54,994,098	62,233,531
mismatches per 100 kbp	527.4	526.21	523.09	517.58
R2				
	CAREx			Konnector2
	strict 0	strict 1	strict 2	
contigs $\geq$ 50,000	133	105	69	72
N50	30,903	27,414	23,492	21,506
misassembled contigs	100	133	197	255
misassembled contigs length	2,971,527	4,253,862	5,136,136	6,069,175
mismatches per 100 kbp	160.85	163.97	164.33	168.76

Tab. 9.7.: A selection of assembly metrics reported by Quast for real datasets R1 and R2 using the *-s* parameter for extended reads.

Table 9.7 shows the selected metrics for the first set of assemblies. On R1, Konnector2 has the greatest N50 value and the smallest number of misassembled contigs as well as the smallest error-rate. CAREx produces more longer contigs and has a smaller total length of misassembled contigs. Since CAREx has more misassemblies but a smaller total misassembled length compared to Konnector2, misassembled contigs for CAREx are shorter than for Konnector2. On R2, CAREx is superior in all five metrics.

Analysis of the second set of assemblies is presented in Table 9.8. Here, Konnector2 achieves a slightly greater N50 value on R1, but is inferior to CAREx in the other metrics, and on R2.

R1				
	CAREx			Konnector2
	strict 0	strict 1	strict 2	
contigs $\geq$ 50,000	670	629	650	633
N50	96,480	114,381	107.806	115.631
misassembled contigs	626	608	638	648
misassembled contigs length	54,227,737	58,604,328	58,014,861	61,500,357
mismatches per 100 kbp	511.03	511.81	511.98	514.07
R2				
	CAREx			Konnector2
	strict 0	strict 1	strict 2	
contigs $\geq$ 50,000	44	28	23	16
N50	17,903	16,500	15,384	13,686
misassembled contigs	195	293	327	474
misassembled contigs length	3,898,745	5,524,109	5,414,538	6,890,369
mismatches per 100 kbp	154.06	155.72	156.91	167.1

Tab. 9.8.: A selection of assembly metrics reported by Quast for real datasets R1 and R2 using the –*merged* parameter for extended reads.

Comparing the different levels of strictness of CAREx, one can see that for R2 the assembly quality decreases with increasing level of strictness for both sets of assemblies, which seems counter-intuitive as the extension error-rate is the greatest for strictness 0. This observation could be explained by the number of produced error-free pseudo-long reads. Although strict mode 0 has a greater error-rate in the connected gap, overall it produces the most error-free connections. For R1, things are not so clear since the assembly quality behaves differently than on R2. This may be caused by the fraction of extended reads with more than two errors which is significantly greater than for R2 (see the unmarked bar areas in Figure 9.2).

To summarize, assemblies produced from reads extended with CAREx are better than those produced with reads from Konnector2 in 17 out of the 20 presented values. However, there is no clear combination of extension strictness and assembly parameters that produces the best values for all five metrics.

9.5 Impact of sequencing errors

In CAREx, sequencing errors in reads may lead to less accurate extensions. The reasons are two-fold. On one hand, erroneous nucleotides can lead to a different consensus sequence in MSAs. On the other hand, the number of candidates may decrease. This is because an erroneous k-mer may contribute to its read's minhash

signature. To evaluate the effect of sequencing errors, we used our error corrector CARE to produce corrected versions of all datasets prior to extension, and computed the edit statistics for those extended corrected versions. Additionally, we extended error-free versions of the simulated datasets to obtain an optimal baseline for extension.

Since the overall impact of error correction is small, we will only give a brief summary of our finding. Table 9.9 lists the average improvements over all standard erroneous datasets. In terms of number of (error-free) connected reads, error correction has similar impact on both simulated datasets and real-world datasets. Error-correction is more beneficial for strict mode 1 and 2 because they require equality of the produced connections on both strands.

	CAREx		
	strict 0	strict 1	strict 2
Simulated			
Number of connected reads (corrected version)	0.84%	2.69%	3.00%
Number of connected reads (error-free version)	0.82%	3.28%	3.75%
Number of error-free gaps (corrected version)	0.03%	2.74%	3.03%
Number of error-free gaps (error-free version)	0.24%	3.40%	3.81%
Real			
Number of connected reads (corrected version)	0.71%	3.07%	3.52%
Number of error-free gaps (corrected version)	0.63%	2.86%	3.24%

Tab. 9.9.: Average improvement of extension quality for corrected datasets over the original datasets.

9.6 Performance

The performance benchmarks of CAREx were conducted on the following system:

M3 (Single-GPU workstation): AMD EPYC 7713P 64-core CPU, 512 GB DDR4 RAM, NVIDIA A100 PCIe GPU with 80 GB HBM2e memory, CUDA Toolkit 11.8

Total program runtime and peak memory consumption for read extension of datasets S2, S3, S4, and R3 are presented in Table 9.10, showing the scaling of runtime with different insert sizes and number of reads. CAREx was run with a memory limit of 200 GB. 48 hash tables were used. CAREx GPU with CPU tables used 20 threads for hashing on the CPU and 2 threads for read extension on the GPU. Strict mode 1 was used. Both tools were run with disabled outward-extension.

On the simulated datasets Konnector2 is the fastest of the CPU-based tools. CAREx (CPU) is up to three times slower. GapFiller does not provide an option for multi-threading which results in a much longer processing time compared to Konnector2 and CAREx (CPU). On real-world dataset R3, CAREx (CPU) is faster than Konnector2. The GPU-accelerated versions of CAREx are significantly faster than the CPU-based tools. Our GPU-based implementation is around seven times faster than our CPU version, allowing for the processing of 600M Human reads of length 148 in a few hours instead of a day. The choice of extension strictness only has little impact on the runtime of CAREx. Using strict mode 0 gives a slightly better performance of around 3% because the comparison of results of different strands is not performed.

Our peak memory usage is reached during hash table construction, when all key-value pairs of all tables are materialized before compacting them into buckets. If the memory limit prohibits the construction of all tables at once, tables are created in batches to reduce the memory consumption which in turn leads to slightly increased construction times. For example, if a memory limit of 480 GB is set for R3, construction times decreases by 9 minutes and 1 minute for (CPU) and (GPU, CPU tables), respectively. The peak memory usage in this case is around 380 GB. However, these gains in runtime are negligible compared to the total runtime.

S2	Threads	Runtime [minutes]	Memory [GB]
CAREx (CPU)	64	43	21
CAREx (GPU, CPU tables)	2+20	8	21 (8)
CAREx (GPU, GPU tables)	2	6	7 (37)
GapFiller	1	3855	4
Konnector2	64	24	11
S3	Threads	Runtime [minutes]	Memory [GB]
CAREx (CPU)	64	99	21
CAREx (GPU, CPU tables)	2+20	19	21 (8)
CAREx (GPU, GPU tables)	2	14	7 (37)
Konnector2	64	62	11
S4	Threads	Runtime [minutes]	Memory [GB]
CAREx (CPU)	64	365	42
CAREx (GPU, CPU tables)	2+20	44	40 (14)
CAREx (GPU, GPU tables)	2	38	14 (72)
Konnector2	64	126	11
R3	Threads	Runtime [hours:minutes]	Memory [GB]
CAREx (CPU)	64	23:18	196
CAREx (GPU, CPU tables)	2+20	3:20	187 (63)
Konnector2	64	27:43	201

Tab. 9.10.: Total program runtime and CPU (GPU) peak memory consumption in GB for the datasets S2, S3, S4, and R3.

Last, we investigated the effect of running CAREx with a reduced number of hash tables with respect to performance and quality of results for dataset R3. Figure 9.3 shows the runtime of CAREx GPU with CPU tables with different numbers of hash tables, and the corresponding amount of produced error-free connections between reads. Similar to the observations made for the error corrector CARE, the number of error-free gaps increases with more hash tables. However, one can also see diminishing returns for greater number of hash tables especially when taking the runtime into account. Achieving the best results requires large computational resources in terms of both processing time and available memory for hash tables.

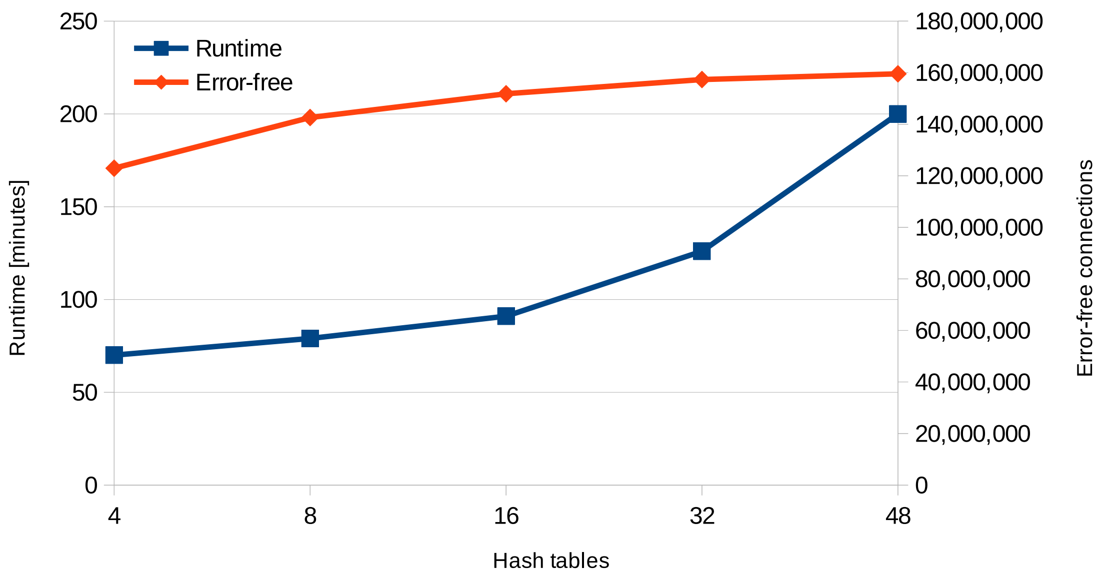


Fig. 9.3.: Runtime and number of error-free connected reads of R3 depending on the number of hash tables.

The current implementation of CAREx does not have proper multi-GPU support. It should scale with the number of GPUs in a similar manner than CARE as it shares many similar components. We assume that CAREx is able to achieve a greater multi-GPU efficiency than CARE with respect to the total program runtime. This is because a sorting step after the extension phase is not mandatory. If no sorted output is requested, results are written to file immediately, concurrently to the ongoing extension phase. This leads to a reduction of sections with serial program execution and thus allows for greater theoretically achievable speedup.

10 CAREx Conclusion

We presented CAREx, a software for the computation of pseudo-long reads from paired-end NGS reads. While the task may be trivial for two overlapping reads, it poses a computational challenge if the insert size is greater than twice the read length. Our MSA-based approach to a targeted local assembly is able to reliably produce such pseudo-long reads for the latter case. In direct comparison to other software, CAREx connects a larger fraction of the input reads. The majority of generated pseudo-long reads is error-free, and their numbers can exceed the total number of connected reads produced by competitors. As can be seen from the results, when a dataset has a large variation in insert size a large fraction of edits is caused by incorrect pseudo-long read lengths. To further improve CAREx, we would need to identify those cases during computation, and either handle them differently, or discard their extension entirely. One possible approach to identification would be to first extend all read pairs to maximum insert size, and then determine whether there are multiple possible positions where the mate could be placed. An important part of CAREx is its efficient parallelization to achieve fast runtimes. It is able to utilize both CPU-sided multi-threading and GPU-enabled acceleration. With full GPU parallelization on a single A100 GPU, CAREx is up to 8 times faster than its competitors.

In the future, we may investigate the use of machine learning techniques such as random forests or deep neural networks. These could improve the selection of nucleotides to append based on the constructed MSA. Similar approaches have been demonstrated to improve the accuracy of variant calling [3]. Other potential research areas involve the use of different types of input datasets. For single-end data one could use two extension tasks instead of four, performing outward-extension beginning at the ends of the read. To support long-read platforms such as PacBio or Oxford Nanopore with their different error models compared to NGS reads, both pair-wise alignments and multiple-sequence alignments need to consider indels. Thus, full semi-global alignments would be required instead of a simple shifted hamming distance. In addition to alignments, hash tables would need to be revised as well. Long-reads would require multi-stage hashing where a long read is split into shorter sections (windows) that are hashed separately. Candidate reads then not only need to have windows with the same hash value as the anchor, but the

corresponding windows also need to be in consistent relative order. The last type of data which may be interesting is metagenomic data. This would require additional preprocessing to separate input reads by species.

Part III

Conclusion

11 Future work

Bioinformatics is a wide field of research. While our general building blocks have been tailored towards our needs for error correction and read extension, they could prove useful in other applications that process sequence data, for example in read mapping. First experiments indicate that our hash tables and alignments can be used to identify exact matches of short NGS reads within a reference genome. Preliminary results show fast execution times on a single GPU for large datasets. It will be interesting to see if we can convert this approach into a fully-functional, GPU-accelerated read mapper.

CARE and CAREx target short NGS reads but their underlying ideas could also be adapted to long third-generation sequencing reads. This would require modifications in all building blocks: Data layout, hashing, and alignments. 1. TGS reads have a larger variation in sequence length. Our current approach of padding all sequences to the maximum length would no longer be viable. Instead, we would need to store sequence without padding and use an additional list of offsets to determine the location of reads within the sequence array. The same applies to quality scores. 2. It is less likely that two long sequences which share a few common k-mers are similar. To identify potentially similar reads, our hashing approach needs to consider hash signatures obtained from multiple, shorter sections of the reads. 3. Pair-wise shifted hamming distance alignments would need to be replaced with slower semi-global alignments. The NVIDIA Hopper GPU architecture introduced DPX instructions that provide hardware-acceleration for common computations in dynamic programming algorithms. These instructions could be utilized for faster semi-global alignments. The construction of multiple-sequence alignments with gaps is more involved since different pair-wise alignments could introduce gaps at different positions within the anchor sequence. Another aspect is the number of reads per dataset. With a fixed dataset coverage, TGS datasets contain less reads than NGS datasets. This may lead to MSAs with a reduced number of rows and would require re-evaluating our thresholds for consensus identification.

Our current implementations of CARE and CAREx already provide good performance. Yet, we see potential for improvement.

In our GPU minhasher, after hash tables have been queried we need to find the set of distinct values per segment which involves sorting of segments of irregular size. The involved sorting and compaction takes at least 20% of the time to find the candidate read ids for a batch of anchor reads. Sorting is a widely used primitive in computer science with ongoing efforts to find faster sorting methods. Recent work [37] suggests the use of bitonic sorting networks and achieves a greater performance than CUB's segmented sort on a wide range of segment sizes. This approach could also be helpful in our use-case.

Another topic is the handling of compressed files. We have seen that the sequence file parsing performance is limited by compute-intensive decompression. The reading and writing of compressed files in CARE and CAREx are based on zlib's `gzRead` and `gzWrite` functions, which are single-threaded. While there exists stand-alone software for multi-threaded gzip compression[1] and decompression[2], their usage before and after our programs would require creating intermediate decompressed files. If we were able to integrate multi-threaded gzip processing directly into our software, similar to RabbitQCPlus2.0, we could provide faster file accesses and reduce disk space usage. In addition to multi-threading, GPU-accelerated sequence file parsing would be beneficial for both compressed and uncompressed files.

Lastly, we may add multi-GPU support to CAREx.

[1] https://zlib.net/pigz/
[2] https://github.com/Piezoid/pugz

12 Conclusion

Error-correction and read extension are an important step in many sequence processing pipelines. In this thesis, we presented two GPU-accelerated applications for those problems, CARE and CAREx, which deliver both a high performance and accurate results and target Illumina NGS reads.

In order to achieve their high accuracy both tools access the full information contained in sequences by constructing multiple-sequence alignments (MSAs) of similar reads to make *context-aware* decisions. This is in contrast to k-mer-based approaches which operate on individual k-mers in isolation, without considering the surrounding positions. To avoid the computational complexity of computing all pair-wise alignments of all reads in a dataset, we developed a minhashing-based index datastructure. After the index is created for a dataset, it can be queried to identify subsets of similar reads. Furthermore, an efficient alignment computation was employed which takes advantage of the properties of the targeted sequencing technology. NGS reads are dominated by substitution errors. This observation is exploited by our developed bit-parallel shifted-hamming-distance alignment that only considers mismatches but not indels. Combined, the index datastructure and the fast pair-wise alignments provide the means for the quick construction of MSAs, a great tool for error correction and read extension. To alleviate computational bottlenecks found in traditional CPU-based applications and to be able to cope with the ever increasing amount of sequencing data, both algorithms are designed for the execution on CUDA-enabled GPUs. While this comes with new challenges in regards of programming, it leads to significantly faster execution times.

Using the MSAs, CARE produces significantly less false-positive corrections than other state-of-the-art software on both real-world datasets and simulated datasets. The number of false positives can be further reduced by employing a Random Forest classifier to make per-base decisions about the incorporation of the MSA consensus nucleotide into the corrected sequence. Our results show that for a Human dataset with 30-fold coverage, CARE is able to remove 90% of sequencing errors in 22 minutes on a modern GPU server using a single NVIDIA A100 GPU. By using multiple GPUs, the runtimes can be further reduced to under ten minutes. CARE achieves a remarkable false-positive rate of $1,500$ per one million corrections

using the Random Forest. Without Random Forest, CARE corrects 86% of all errors, with a false-positive rate of $1,800$. The competitors, which are exclusively CPU-based programs, fall short in at least one of: percentage of corrected errors, false-positive rate, and runtime. For example, BFC correctly identifies 92% of the errors in 90 minutes with 13 times more false-positive corrections than CARE with Random Forest. Karect produces the second-fewest false-positives which are still 3.5 times larger than CARE's, achieves only 82% corrections and takes several days to complete on our machine. Lighter is the fastest tool, taking only 45 minutes. However, it produces the greatest false-positive rate ($74,000$ per million corrections) and handles only 79% of all errors. In the performance evaluation of CARE, we found that file accesses can limit the performance. Those cannot be easily improved via parallelization since the total transfer rate between storage device and RAM is limited. In our benchmarks, we observed runtime improvements of up to 30% when using a PCIe SSD instead of a HDD. The IO limitations can become even more prominent when multiple GPUs are used. The fraction of IO in the total execution time is expected to increase further with faster GPUs.

CAREx utilizes the MSAs to compute more error-free connections of read pairs than its competitors. The GPU-accelerated version of CAREx takes 3.3 hours on an A100 GPU to process a Human dataset with 30-fold coverage. Our algorithm is able to connect between 50% and 60% of all read pairs without errors. In contrast, Konnector2 achieves only 35% error-free connected read pairs with a runtime of 27.5 hours. The quality of our results depend on the level of strictness when comparing temporary results produced on different strands. More strict extension leads to an increased accuracy, but reduces the overall number of generated pseudo-long read.

Although GPU-acceleration yields superior runtimes, both CARE and CAREx offer CPU-based code paths at reduced speeds. Thus, both programs are also able to deliver the same accurate results on work-stations that are not equipped with a CUDA-enabled GPU. This makes them widely usable. On a 64-core CPU, the CPU variants of CARE and CAREx are around $4 - 8$ times slower than the respective GPU variants on a single A100 GPU.

To allow simple access to our software, and to enable other interested researchers to use and improve our findings, CARE and CAREx are licensed under GNU General Public License v3 and are publicly available on Github under `https://github.com/fkallen/CARE` and `https://github.com/fkallen/CAREx`.

Bibliography

[1]US DOE Joint Genome Institute: Hawkins Trevor 4 Branscomb Elbert 4 Predki Paul 4 Richardson Paul 4 Wenning Sarah 4 Slezak Tom 4 Doggett Norman 4 Cheng Jan-Fang 4 Olsen Anne 4 Lucas Susan 4 Elkin Christopher 4 Uberbacher Edward 4 Frazier Marvin 4, RIKEN Genomic Sciences Center: Sakaki Yoshiyuki 9 Fujiyama Asao 9 Hattori Masahira 9 Yada Tetsushi 9 Toyoda Atsushi 9 Itoh Takehiko 9 Kawagoe Chiharu 9 Watanabe Hidemi 9 Totoki Yasushi 9 Taylor Todd 9, Genoscope, et al. "Initial sequencing and analysis of the human genome". In: *nature* 409.6822 (2001), pp. 860–921 (cit. on p. 1).

[2]Amin Allam, Panos Kalnis, and Victor Solovyev. "Karect: accurate correction of substitution, insertion and deletion errors for next-generation sequencing data". In: *Bioinf.* 31.21 (2015) (cit. on pp. 17, 26).

[3]Gunjan Baid, Daniel E Cook, Kishwar Shafin, et al. "DeepConsensus improves the accuracy of sequences with a gap-aware sequence transformer". In: *Nature Biotechnology* (2022), pp. 1–7 (cit. on p. 113).

[4]Anton Bankevich, Sergey Nurk, Dmitry Antipov, et al. "SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing". In: *J. Comput. Biol.* 19.5 (2012) (cit. on pp. 17, 60).

[5]David R. Bentley, Shankar Balasubramanian, Harold P. Swerdlow, et al. "Accurate whole human genome sequencing using reversible terminator chemistry". In: *Nature* 456.7218 (Nov. 2008), pp. 53–59 (cit. on p. 29).

[6]K Berlin, Sergey Koren, Chen-Shan Chin, et al. "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing". In: *Nat Biotech* 33 (2015) (cit. on p. 19).

[7]Marten Boetzer and Walter Pirovano. "Toward almost closed genomes with GapFiller". In: *Genome Biology* 13.6 (June 2012), R56 (cit. on p. 90).

[8]Paola Bonizzoni and Gianluca Della Vedova. "The complexity of multiple sequence alignment with SP-score that is a metric". In: *Theoretical Computer Science* 259.1 (2001), pp. 63–79 (cit. on p. 10).

[9]Andrei Z Broder. "Identifying and filtering near-duplicate documents". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2000 (cit. on p. 19).

[10]Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, et al. "Phased diploid genome assembly with single-molecule real-time sequencing". In: *Nature methods* 13.12 (2016), pp. 1050–1054 (cit. on p. 18).

[11]Francis S Collins, Eric D Green, Alan E Guttmacher, Mark S Guyer, and US National Human Genome Research Institute. “A vision for the future of genomics research”. In: *nature* 422.6934 (2003), pp. 835–847 (cit. on p. 1).

[12]Petr Danecek, James K Bonfield, Jennifer Liddle, et al. “Twelve years of SAMtools and BCFtools”. In: *GigaScience* 10.2 (Feb. 2021). giab008. eprint: `https://academic.oup.com/gigascience/article-pdf/10/2/giab008/36332246/giab008.pdf` (cit. on p. 106).

[13]Maciej Długosz and Sebastian Deorowicz. “RECKONER: read error corrector based on KMC”. In: *Bioinf.* 33.7 (2017) (cit. on pp. 17, 25).

[14]Dent Earl, Keith Bradnam, John St. John, et al. “Assemblathon 1: A competitive assessment of de novo short read assembly methods”. In: *Genome Res.* 21 (2011) (cit. on p. 60).

[15]B Ewing, L Hillier, M C Wendl, and P Green. “Base-calling of automated sequencer traces using phred. I. Accuracy assessment”. en. In: *Genome Res* 8.3 (Mar. 1998), pp. 175–185 (cit. on p. 5).

[16]Irena Fischer-Hwang, Idoia Ochoa, Tsachy Weissman, et al. “Denoising of Aligned Genomic Data”. In: *Scientific reports* 9.1 (2019) (cit. on p. 23).

[17]Paul Greenfield, Konsta Duesing, Alexie Papanicolaou, et al. “Blue: correcting sequencing errors using consensus and context”. In: *Bioinf.* 30.19 (2014) (cit. on pp. 17, 25).

[18]Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, et al. “QUAST: quality assessment tool for genome assemblies”. In: *Bioinf.* 29.8 (2013) (cit. on p. 60).

[19]Dan Gusfield. “Algorithms on stings, trees, and sequences: Computer science and computational biology”. In: *Acm Sigact News* 28.4 (1997) (cit. on p. 10).

[20]Yun Heo, Anand Ramachandran, Wen-Mei Hwu, et al. “BLESS 2: accurate, memory-efficient and fast error correction method”. In: *Bioinf.* 32.15 (2016) (cit. on pp. 17, 25).

[21]Mahdi Heydari, Giles Miclotte, Piet Demeester, et al. “Evaluation of the impact of Illumina error correction tools on de novo genome assembly”. In: *BMC Bioinform.* 18.1 (2017) (cit. on p. 23).

[22]Mahdi Heydari, Giles Miclotte, Yves Van de Peer, et al. “Illumina error correction near highly repetitive DNA regions improves de novo genome assembly”. In: *BMC Bioinform.* 20.1 (2019) (cit. on p. 26).

[23]Weichun Huang, Leping Li, Jason R Myers, et al. “ART: a next-generation sequencing read simulator”. In: *Bioinf.* 28.4 (2012) (cit. on p. 49).

[24]Che-Lun Hung, Yu-Shiang Lin, Chun-Yuan Lin, Yeh-Ching Chung, and Yi-Fang Chung. “CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on Multi-GPUs”. In: *Computational Biology and Chemistry* 58 (Oct. 2015), pp. 62–68 (cit. on p. 20).

[25]L Ilie and M Molnar. "RACER: Rapid and accurate correction of errors in reads". In: *Bioinf.* 29.19 (2013) (cit. on pp. 17, 25).

[26]Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. "HiTEC: accurate error correction in high-throughput sequencing data". en. In: *Bioinformatics* 27.3 (Nov. 2010), pp. 295–302 (cit. on p. 17).

[28]Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, et al. "ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter". en. In: *Genome Res* 27.5 (Feb. 2017), pp. 768–777 (cit. on p. 90).

[29]Yanrong Ji, Zhihan Zhou, Han Liu, and Ramana V Davuluri. "DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome". In: *Bioinformatics* 37.15 (Feb. 2021), pp. 2112–2120. eprint: `https://academic.oup.com/bioinformatics/article-pdf/37/15/2112/50927437/btab083.pdf` (cit. on p. 18).

[30]John Jumper, Richard Evans, Alexander Pritzel, et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (July 2021), pp. 583–589 (cit. on p. 18).

[32]Daniel Jünger, Robin Kobus, André Müller, et al. "WarpCore: A Library for fast Hash Tables on GPUs". In: *HiPC 2020*. IEEE, 2020, pp. 11–20 (cit. on p. 39).

[33]Felix Kallenborn, Julian Cascitti, and Bertil Schmidt. "CARE 2.0: reducing false-positive sequencing error corrections using machine learning". In: *BMC Bioinformatics* 23.1 (June 2022) (cit. on p. 3).

[34]Felix Kallenborn, Andreas Hildebrandt, and Bertil Schmidt. "CARE: context-aware sequencing read error correction". In: *Bioinformatics* 37.7 (Aug. 2020), pp. 889–895 (cit. on p. 3).

[35]Wei-Chun Kao, Andrew H Chan, and Yun S Song. "ECHO: a reference-free short-read error correction algorithm". In: *Genome research* 21.7 (2011) (cit. on pp. 17, 26).

[36]Robin Kobus, André Müller, Daniel Jünger, Christian Hundt, and Bertil Schmidt. "MetaCache-GPU: ultra-fast metagenomic classification". In: *Proceedings of the 50th International Conference on Parallel Processing*. 2021, pp. 1–11 (cit. on p. 20).

[37]Robin Kobus, Johannes Nelgen, Valentin Henkys, and Bertil Schmidt. "Faster Segmented Sort on GPUs". In: *Euro-Par 2023: Parallel Processing*. Ed. by José Cano, Marios D. Dikaiakos, George A. Papadopoulos, Miquel Pericàs, and Rizos Sakellariou. Cham: Springer Nature Switzerland, 2023, pp. 664–678 (cit. on p. 118).

[38]David Laehnemann, Arndt Borkhardt, and Alice Carolyn McHardy. "Denoising DNA deep sequencing data—high-throughput sequencing errors and their correction". In: *Briefings in bioinformatics* 17.1 (2016), pp. 154–179 (cit. on p. 17).

[39]H Li. "BFC: correcting Illumina sequencing errors". In: *Bioinf.* 31.17 (Sept. 2015) (cit. on pp. 17, 25).

[40] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform". In: *Bioinformatics* 25.14 (May 2009), pp. 1754–1760. eprint: https://academic.oup.com/bioinformatics/article-pdf/25/14/1754/605544/btp324.pdf (cit. on p. 106).

[41] Qing Li, Weidong Cai, Xiaogang Wang, et al. "Medical image classification with convolutional neural network". In: *2014 13th International Conference on Control Automation Robotics and Vision (ICARCV)*. IEEE, Dec. 2014 (cit. on p. 18).

[42] A Limasset, JF Flot, and P Peterlongo. "Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs." In: *Bioinf.* (2019) (cit. on pp. 17, 26).

[43] Binghang Liu, Jianying Yuan, Siu-Ming Yiu, et al. "COPE: an accurate k-mer-based pair-end reads connection tool to facilitate genome assembly". In: *Bioinformatics* 28.22 (Oct. 2012), pp. 2870–2874. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/22/2870/671354/bts563.pdf (cit. on p. 90).

[44] Chi-Man Liu, Thomas Wong, Edward Wu, et al. "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads". In: *Bioinformatics* 28.6 (2012), pp. 878–879 (cit. on p. 20).

[45] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. "GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment". In: *High Performance Computing-HiPC 2006: 13th International Conference, Bangalore, India, December 18-21, 2006. Proceedings 13*. Springer. 2006, pp. 363–374 (cit. on p. 20).

[46] Y Liu, J Schröder, and B Schmidt. "Musket: a multistage *k*-mer spectrum-based error corrector for Illumina sequence data". In: *Bioinf.* 29.3 (2013) (cit. on pp. 17, 25).

[47] Yongchao Liu and Bertil Schmidt. "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing". In: *IEEE Design & Test* 31.1 (2013), pp. 31–39 (cit. on p. 20).

[48] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI". In: *BMC Bioinformatics* 12.1 (Mar. 2011), p. 85 (cit. on p. 17).

[49] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC bioinformatics* 14 (2013), pp. 1–10 (cit. on p. 20).

[50] Ruibang Luo, Binghang Liu, Yinlong Xie, et al. "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler". In: *GigaScience* 1.1 (Dec. 2012). 2047-217X-1-18. eprint: https://academic.oup.com/gigascience/article-pdf/1/1/2047-217X-1-18/25510880/13742_2012_article_18.pdf (cit. on pp. 17, 90).

[51] Tanja Magoč and Steven L. Salzberg. "FLASH: fast length adjustment of short reads to improve genome assemblies". In: *Bioinformatics* 27.21 (Sept. 2011), pp. 2957–2963. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/21/2957/576912/btr507.pdf (cit. on p. 90).

[52] Medhat Mahmoud, Nastassia Gobet, Diana Ivette Cruz-Dávalos, et al. “Structural variant calling: the long and the short of it”. In: *Genome biology* 20.1 (2019), pp. 1–14 (cit. on p. 19).

[53] Guillaume Marcais and Carl Kingsford. “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”. In: *Bioinf.* 27.6 (2011) (cit. on p. 59).

[54] André Müller, Christian Hundt, Andreas Hildebrandt, et al. “MetaCache: context-aware classification of metagenomic reads using minhashing”. In: *Bioinf.* 33.23 (2017) (cit. on p. 19).

[55] Francesca Nadalin, Francesco Vezzi, and Alberto Policriti. “GapFiller: a de novo assembly approach to fill the gap within paired reads”. In: *BMC Bioinformatics* 13.14 (Sept. 2012), S8 (cit. on pp. 19, 90).

[58] B D Ondov, Todd J. Treangen, Páll Melsted, et al. “Mash: fast genome and metagenome distance estimation using MinHash”. In: *Genome Biology* 17:132 (2016) (cit. on p. 19).

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 45).

[60] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. “An Eulerian path approach to DNA fragment assembly”. In: *Proceedings of the National Academy of Sciences* 98.17 (Aug. 2001), pp. 9748–9753 (cit. on p. 17).

[61] V Popic and S Batzoglou. “Privacy-Preserving Read Mapping Using Locality Sensitive Hashing and Secure Kmer Voting”. In: *bioRxiv* (2016) (cit. on p. 19).

[62] Ryan Poplin, Pi-Chuan Chang, David Alexander, et al. “A universal SNP and small-indel variant caller using deep neural networks”. In: *Nature Biotechnology* 36.10 (Sept. 2018), pp. 983–987 (cit. on p. 20).

[63] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. “Reducing storage requirements for biological sequence comparison”. In: *Bioinformatics* 20.18 (2004), pp. 3363–3369 (cit. on p. 19).

[64] Leena Salmela and Eric Rivals. “LoRDEC: accurate and efficient long read error correction”. In: *Bioinformatics* 30.24 (2014), pp. 3506–3514 (cit. on p. 17).

[65] Leena Salmela and Jan Schröder. “Correcting errors in short reads by multiple alignments”. In: *Bioinf.* 27.11 (2011) (cit. on pp. 17, 26).

[66] F Sanger, S Nicklen, and A R Coulson. “DNA sequencing with chain-terminating inhibitors”. en. In: *Proc. Natl. Acad. Sci. U. S. A.* 74.12 (Dec. 1977), pp. 5463–5467 (cit. on p. 5).

[67] Melanie Schirmer, Rosalinda D’Amore, and Umer Z .and others Ijaz. “Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data”. In: *BMC Bioinform.* 17.1 (2016) (cit. on p. 29).

[68] Bertil Schmidt and Andreas Hildebrandt. "Next-generation sequencing: big data meets high performance computing". en. In: *Drug Discov Today* 22.4 (Feb. 2017), pp. 712–717 (cit. on p. 1).

[69] Marcel H Schulz, David Weese, Manuel Holtgrewe, et al. "Fiona: a parallel and automatic strategy for read error correction". In: *Bioinf.* 30.17 (2014) (cit. on pp. 17, 26).

[70] Milan Shah, Reece Neff, Hancheng Wu, et al. "Accelerating Random Forest Classification on GPU and FPGA". In: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP '22. Bordeaux, France: Association for Computing Machinery, 2023 (cit. on p. 45).

[71] Atul Sharma, Pranjal Jain, Ashraf Mahgoub, et al. "Lerna: transformer architectures for configuring error correction tools for short- and long-read genome sequencing". In: *BMC Bioinformatics* 23.1 (Jan. 2022), p. 25 (cit. on p. 18).

[72] Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Müller-Wittig. "A Parallel Algorithm for Error Correction in High-Throughput Short-Read Data on CUDA-Enabled Graphics Hardware". In: *Journal of Computational Biology* 17.4 (Apr. 2010), pp. 603–615 (cit. on p. 18).

[73] David H. Silver, Shay Ben-Elazar, Alexei Bogoslavsky, and Itai Yanai. "ELOPER: elongation of paired-end reads as a pre-processing tool for improved de novo genome assembly". In: *Bioinformatics* 29.11 (Apr. 2013), pp. 1455–1457. eprint: `https://academic.oup.com/bioinformatics/article-pdf/29/11/1455/17100069/btt169.pdf` (cit. on pp. 19, 90).

[74] Mikang Sim, Jongin Lee, Suyeon Wy, et al. "Generation and application of pseudo–long reads for metagenome assembly". In: *GigaScience* 11 (2022) (cit. on p. 90).

[75] Jared T Simpson and Richard Durbin. "Efficient de novo assembly of large genomes using compressed data structures". In: *Genome research* 22.3 (2012) (cit. on pp. 17, 25).

[76] L Song, L Florea, and B Langmead. "Lighter: fast and memory-efficient sequencing error correction without counting". In: *Genome Biology* 15.11 (2014) (cit. on pp. 17, 25).

[77] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, et al. "Big data: astronomical or genomical?" In: *PLoS biology* 13.7 (2015), e1002195 (cit. on p. 1).

[78] Julie D. Thompson, Toby. J. Gibson, and Des G. Higgins. "Multiple Sequence Alignment Using ClustalW and ClustalX". In: *Current Protocols in Bioinformatics* 00.1 (Aug. 2002) (cit. on p. 20).

[79] Benjamin P. Vandervalk, Chen Yang, Zhuyi Xue, et al. "Konnector v2.0: pseudo-long reads from paired-end sequencing data". In: *BMC Medical Genomics* 8.3 (Sept. 2015), S1 (cit. on pp. 19, 90).

[80] Jeremy R Wang, James Holt, Leonard McMillan, and Corbin D Jones. "FMLRC: Hybrid long read error correction using an FM-index". In: *BMC bioinformatics* 19 (2018), pp. 1–11 (cit. on p. 17).

[82] Lifeng Yan, Zekun Yin, Hao Zhang, et al. "RabbitQCPlus 2.0: More Efficient and Versatile Quality Control for Sequencing Data". In: *Methods* (2023) (cit. on pp. 3, 85).

[83] Jiajie Zhang, Kassian Kobert, Tomáš Flouri, and Alexandros Stamatakis. "PEAR: a fast and accurate Illumina Paired-End reAd mergeR". In: *Bioinformatics* 30.5 (2014), pp. 614–620 (cit. on p. 90).

[84] Aleksey V. Zimin, Guillaume Marçais, Daniela Puiu, et al. "The MaSuRCA genome assembler". In: *Bioinformatics* 29.21 (Aug. 2013), pp. 2669–2677. eprint: https://academic.oup.com/bioinformatics/article-pdf/29/21/2669/18533361/btt476.pdf (cit. on pp. 17, 90).

Webpages

[@27] Illumina. *Understanding Illumina Quality Scores*. 2014. URL: https://www.illumina.com/content/dam/illumina-marketing/documents/products/technotes/technote_understanding_quality_scores.pdf (visited on Oct. 18, 2022) (cit. on p. 36).

[@31] *June 2000 White House Event*. 2000. URL: https://www.genome.gov/10001356/june-2000-white-house-event (visited on Aug. 21, 2023) (cit. on p. 1).

[@56] NVIDIA. *CUDA C++ Programming Guide*. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on Aug. 21, 2023) (cit. on p. 12).

[@57] NVIDIA. *NVIDIA Clara for Genomics*. 2023. URL: https://www.nvidia.com/en-us/clara/genomics/ (visited on Aug. 21, 2023) (cit. on p. 20).

[@81] Kris A. Wetterstrand. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)*. 2023. URL: https://www.genome.gov/sequencingcostsdata (visited on Aug. 21, 2023) (cit. on p. 2).

List of Figures

1.1. Sequencing costs per human genome as reported by the National Human Genome Research Institute [@81] 2

2.1. Paired-end sequencing of a DNA fragment. Typically, not the full fragment length is sequenced per strand. 6

2.2. An MSA of five sequences with its consensus string and its probability profile. 11

2.3. A block-wide parallel reduction 15

4.1. Left: k-mer frequency histogram. A frequency threshold separates weak k-mers (yellow) from solid k-mers (green). Right: k-mer based modification of a sequence. 26

4.2. Workflow of CARE: (a) The signature of an anchor read (r_i) is determined by minhashing and used to query the precomputed hash tables. The retrieved reads form the candidate read set $C(r_i)$. (b) All reads in $C(r_i)$ are aligned to r_i. Reads with a relatively low semi-global pair-wise alignment quality are removed, resulting in the filtered set of candidate reads ($F(r_i)$). (c) The initial MSA is constructed around the center r_i using $F(r_i)$. The MSA is refined by removing candidate reads with a significantly different pattern from the anchor (i.e. r_{15}, r_{22}, r_7 in the example). (d) The anchor read (the seventh nucleotide in r_i in the example) and optionally some of the candidates are corrected (the fifth nucleotide in r_2 in the example), using a provided random forest trained for correction. 28

4.3. Constructing a look-up table from the generated key-value pairs of a single hash function. 37

4.4. Producer-consumer-pipeline for CPU hash tables (left) and a simple pipeline for GPU hash tables (right). Blue and green indicate CPU workloads and GPU workloads, respectively. 41

4.5. Finding candidate read ids for a batch of two reads. Three hash functions are used. Different colors indicate results of different hash functions. . 42

4.6. Bit-parallel hamming distance computation. Mismatching characters are highlighted red. 43

4.7. An array with 16 elements is evenly distributed between two GPUs. The array is collectively accessed by each GPU via index list. Indices per GPU are identified via (non-stable) multi-split and exchanged. Locally gathered data is sent back to the respective GPUs. The second all-to-all operation includes a reordering of gathered data to match the order of input indices. 47

5.1. Variation of correction parameters. 53
5.2. TP depending on random forest usage and candidate correction. 55
5.3. FP depending on random forest usage and candidate correction. 55
5.4. Average TP ratio and FP ratio over CARE RF on simulated HiSeq data. Greater numbers are better for TP. Smaller numbers are better for FP. . 57
5.5. Average TP ratio and FP ratio over CARE RF on simulated MiSeq data. Greater numbers are better for TP. Smaller numbers are better for FP. . 58

6.1. Speed-up of hash table construction. 63
6.2. Relative time spent in the different steps during correction phase. . . . 64
6.3. Speed-up over the single-threaded CPU version in the correction phase. 65
6.4. Total speed-up of GPU version over CPU version. 66
6.5. Relative time spent in the different steps during correction with random forests. 67
6.6. Speed-up of GPU version over CPU version in correction phase with random forests. 68
6.7. Total speed-up of GPU version over CPU version with random forests. . 68
6.8. Runtime of the correction phase with GPU hash tables depending on read data location. 8-bit quality scores are used. 69
6.9. Runtime of the correction phase with GPU hash tables depending on read data location. 2-bit quality scores are used. 70
6.10. Runtime of the correction phase with CPU hash tables depending on read data location. 8-bit quality scores are used. 71
6.11. Runtime of the correction phase with CPU hash tables depending on read data location. 2-bit quality scores are used. 71
6.12. Multi-GPU speedup of the correction phase of R2 with GPU hash tables for different data distribution schemes. A single thread is used. 75
6.13. Timeline to determine the candidate lists per anchor. 76
6.14. Timeline to correct the anchors. 76
6.15. Multi-GPU hash table access with all-to-all copies using CUDA API calls. 78
6.16. Multi-GPU hash table access with all-to-all copies using copy kernels. . 78

8.1. Layout of the four sequences for extension tasks. 91

8.2. Five different outcomes of extension task processing. Mismatches between the two strands (i.e. non-complementary bases) are marked in red. The right-hand table indicates whether a pseudo-long read can be constructed for a specific strictness. 95

9.1. Percentage of connected reads for simulated datasets. GapFiller did not finish for S3, S4, S6, and S7. 102
9.2. Percentage of connected read pairs of real-world datasets. Error-rate is given at the top of each bar. 106
9.3. Runtime and number of error-free connected reads of R3 depending on the number of hash tables. 112

List of Tables

3.1. Selection of error correction tools for NGS data and TGS data with their type of corrected errors and the key utilized data-structure: Bloom filter (BF), Hash table (HT), Suffix Array (SA), FM-index, and De Bruijn graph (DBG). 17

4.1. Features extracted from an MSA. c is the estimated dataset coverage. x is the consensus nucleotide. o is the original nucleotide of the sequence to be corrected (anchor or candidate). Features 6-14 describe only the currently inspected column, whereas features 1-5 are properties that cover multiple columns and are constant for all positions of the same anchor or the same candidate, respectively. 45

5.1. Simulated datasets of collections A-D with their respective number of reads. Dataset number 4 is only available in collection A. 51

5.2. Collection R of real-world HiSeq datasets. 51

5.3. Lost 21-mers for dataset R1. 59

5.4. Selected assembly results for dataset R3. 60

6.1. Runtime in seconds to count the number of reads per file. 62

6.2. Scaling of readstorage construction with the number of encoder-threads. Runtime is given in seconds. The input file is already cached in RAM. . 62

6.3. GPU memory usage [MB] during correction phase. 65

6.4. Total runtime of correction of dataset A4. Runtime is given in minutes. 72

6.5. Comparison of the improved versions on M2. The single-GPU version uses CPU hash tables. Times are given in minutes:seconds. 79

6.6. Runtime in seconds for GPU minhasher construction with multiple GPUs. The construction of CPU hash tables with a single GPU takes 220 seconds. 79

6.7. Best configuration depending on the number of GPUs. (A) CPU version – 64 threads, (B) GPU version with CPU hash tables and replicated reads – 2 consumers and 20 producers per GPU, (C) GPU version with GPU hash tables and distributed reads – 4 threads, (D) GPU version with GPU hash tables and replicated reads – 4 threads. Runtimes are given in [minutes:seconds]. 80

6.8. Speedup of GPU-accelerated FASTQ parsing when the parsed data should be stored in host memory. 83
6.9. Speedup of GPU-accelerated FASTQ parsing when the parsed data should be stored in device memory. 83

9.1. Simulated (S1-S7) and real-world (R1-R3) datasets used for evaluation. 100
9.2. Total error rate of filled gaps for simulated reads. 103
9.3. Difference between expected gap size and pseudo-long read gap size for dataset S3. 103
9.4. Total error rate of filled gaps excluding length differences. 104
9.5. Pseudo-long read lengths when outward extension is enabled. Strict mode 0 was used. 105
9.6. Error-rate of outward extended regions. 105
9.7. A selection of assembly metrics reported by Quast for real datasets R1 and R2 using the *-s* parameter for extended reads. 108
9.8. A selection of assembly metrics reported by Quast for real datasets R1 and R2 using the *–merged* parameter for extended reads. 109
9.9. Average improvement of extension quality for corrected datasets over the original datasets. 110
9.10. Total program runtime and CPU (GPU) peak memory consumption in GB for the datasets S2, S3, S4, and R3. 111

A.1. Results for A1. 135
A.2. Results for A2. 136
A.3. Results for A3. 136
A.4. Results for A4. 137
A.5. Lost 21-mers for dataset R1. 137
A.6. Lost 21-mers for dataset R2. 138
A.7. Lost 21-mers for dataset R3. 138
A.8. Selected assembly results for dataset R1. 139
A.9. Selected assembly results for dataset R2. 139
A.10. Selected assembly results for dataset R3. 140

Appendix A

A.1 CARE: Results for simulated data

A1	CARE	CARE+RF	BFC	Musket
TP	27.6M	28.2M	28.7M	27.4M
FP	6,972	4,644	164,187	304,402
FN	1.6M	0.9M	0.4M	1.7M
TN	3.0G	3.0G	3.0G	3.0G
FPR	252.92	164.22	5,684.37	11,003.54
Sensitivity	0.95	0.97	0.99	0.94
Specificity	1	1	1	1
Precision	1	1	0.99	0.99
	SGA	BCOOL	Lighter	Karect
TP	28.1M	28.1M	27.8M	28.5M
FP	163,617	143,866	405,593	82,652
FN	1.0M	1.0M	1.4M	0.6M
TN	3.0G	3.0G	3.0G	3.0G
FPR	5,791.33	5,097.88	14,400.64	2,889.87
Sensitivity	0.96	0.96	0.95	0.98
Specificity	1	1	1	1.00
Precision	0.99	0.99	0.99	1

Tab. A.1.: Results for A1.

A2	CARE	CARE+RF	BFC	Musket
TP	33.6M	34.0M	34.6M	33.9M
FP	5,249	2,129	51,866	102,655
FN	1.6M	1.0M	0.4M	1.0M
TN	3.6G	3.6G	3.6G	3.6G
FPR	109.83	62.69	1,498.06	3,018.26
Sensitivity	0.95	0.97	0.99	0.97
Specificity	1	1	1	1
Precision	1	1	1	1
	SGA	BCOOL	Lighter	Karect
TP	33.7M	34.0M	34.0M	34.4M
FP	42,008	46,499	124,522	49,202
FN	1.2M	0.9M	1.0M	0.5M
TN	3.6G	3.6G	3.6G	3.6G
FPR	1,244.78	1,364.57	3,650.62	1,426.32
Sensitivity	0.96	0.97	0.97	0.99
Specificity	1	1	1	1.00
Precision	1	1	1	1

Tab. A.2.: Results for A2.

A3	CARE	CARE+RF	BFC	Musket
TP	23.6	24.5M	24.7M	22.5M
FP	10,646	7,641	377,975	493,773
FN	2.1M	1.2M	0.9M	3.1M
TN	2.6G	2.6G	2.6G	2.6G
FPR	451.7	312.07	15,043.12	21,463.34
Sensitivity	0.92	0.95	0.96	0.88
Specificity	1	1	1	1
Precision	1	1	0.98	0.98
	SGA	BCOOL	Lighter	Karect
TP	24.1M	23.8M	23.2M	23.9M
FP	597,770	627,532	888,141	159,309
FN	1.6M	1.8M	2.5M	1.7M
TN	2.6G	2.6G	2.6G	2.6G
FPR	24,238.49	25,668.63	36,942.57	6,611.58
Sensitivity	0.94	0.93	0.9	0.93
Specificity	1	1	1	1.00
Precision	0.98	0.97	0.96	0.99

Tab. A.3.: Results for A3.

A4	CARE	CARE+RF	BFC	Musket
TP	760.1M	801.4M	814.6M	647.6M
FP	1.4M	1.2M	17.0M	34.6M
FN	125.6M	84.3M	71.2M	238.1M
TN	90.6	90.6G	90.6G	90.6G
FPR	1,836.69	1,546.08	20,413.46	50,765.89
Sensitivity	0.86	0.9	0.92	0.73
Specificity	1	1	1	1
Precision	1	1	0.98	0.95
	SGA	BCOOL	Lighter	Karect
TP	776.7M	768.7M	696.2M	725.5M
FP	34.1M	39.6M	55.7M	3.9M
FN	109.1M	117.0M	189.6M	160.2M
TN	90.6G	90.6G	90.6G	90.6G
FPR	42,091.74	49,019.64	74,086.65	5,339.75
Sensitivity	0.88	0.87	0.79	0.82
Specificity	1	1	1	1.00
Precision	0.96	0.95	0.93	0.99

Tab. A.4.: Results for A4.

A.2 CARE: K-mer evaluation results for real-world data

Coverage	CARE	CARE+RF	BFC	Musket	SGA	BCOOL	Lighter	Karect
1	3,700	2,888	11,496	141,236	5,567	6,675	44,807	17,682
2	328	247	4,710	110,990	1,309	3,045	23,429	7,092
3	73	39	2,928	94,440	135	1,902	15,917	4,764
4	59	9	845	84,262	25	1,449	10,703	3,265
5	14	2	172	74,006	3	1,368	6,985	2,298
6	10	1	41	60,491	3	1,387	4,495	1,582
7	21	0	29	42,896	0	1,237	2,476	919
8	4	0	4	19,948	2	1,307	1,305	544
9	11	0	1	476	0	1,447	644	203
10	21	0	1	83	0	1,686	294	48
Sum	4,241	3,186	20,227	628,828	7,044	21,503	111,055	38,397

Tab. A.5.: Lost 21-mers for dataset R1.

Coverage	CARE	CARE+RF	BFC	Musket	SGA	BCOOL	Lighter	Karect
1	137,517	121,004	187,469	195,864	161,570	158,568	179,773	209,077
2	55,260	35,734	110,031	130,634	64,626	92,080	112,228	131,845
3	35,976	15,523	89,729	136,256	11,182	92,686	111,140	127,533
4	21,704	7,275	45,748	143,262	1,167	96,251	104,020	126,690
5	10,571	2,377	9,599	143,747	133	94,475	83,661	118,392
6	4,134	477	1,353	141,857	46	93,473	59,933	104,004
7	1,668	76	175	128,612	12	92,133	38,153	74,085
8	569	6	90	95,111	9	88,405	22,421	37,102
9	163	15	44	44,584	8	87,783	11,898	10,431
10	134	0	21	574	1	86,811	5,825	599
Sum	267,696	182,487	444,259	1,160,501	238,754	982,665	729,052	939,758

Tab. A.6.: Lost 21-mers for dataset R2.

Coverage	CARE	CARE+RF	BFC	Musket	SGA	BCOOL	Lighter	Karect
1	20,833	20,072	29,126	36,154	23,732	21,967	31,698	34,237
2	2,520	2,369	7,164	12,853	4,391	4,197	8,647	9,076
3	538	537	3,939	9,793	926	2,219	5,529	5,471
4	199	162	1,996	9,541	241	2,257	4,265	4,486
5	80	68	636	10,397	84	3,199	3,622	3,956
6	23	23	141	9,059	42	5,439	2,539	2,952
7	18	25	50	4,773	30	7,679	1,456	1,157
8	6	4	21	81	5	10,763	769	284
9	1	3	4	13	3	14,377	349	108
10	0	2	4	3	1	18,434	109	64
Sum	24,218	23,265	43,081	92,667	29,455	90,531	58,983	61,791

Tab. A.7.: Lost 21-mers for dataset R3.

A.3 CARE: Assembly results for real-world data

R1	Unprocessed	CARE	CARE+RF	BFC	Musket
# contigs $\geq 50k$	64	65	63	59	42
# contigs	26,592	26,599	26,573	26,787	31,048
# misassembled	1,219	1,239	1,205	1,260	1,930
N50	7,659	7,660	7,696	7,607	6,126
NG50	8,518	8,537	8,575	8,497	6,635
	SGA	BCOOL	Lighter	Karect	
# contigs $\geq 50k$	64	65	54	55	
# contigs	26,602	27,124	27,551	27,198	
# misassembled	1,217	1,639	1,349	1,289	
N50	7,660	7,436	7,378	7,464	
NG50	8,513	8,189	8,108	8,220	

Tab. A.8.: Selected assembly results for dataset R1.

R2	Unprocessed	CARE	CARE+RF	BFC	Musket
# contigs $\geq 50k$	575	587	587	561	456
# contigs	9,227	9,154	8,983	9,455	10,748
# misassembled	670	663	669	724	681
N50	43,681	44,189	45,909	42,343	33,564
NG50	43,568	44,165	45,807	42,281	33,375
	SGA	BCOOL	Lighter	Karect	
# contigs $\geq 50k$	570	626	487	483	
# contigs	9,449	8,354	10,901	10,811	
# misassembled	666	689	688	676	
N50	42,514	50,620	33,426	34,551	
NG50	42,350	50,280	33,301	34,203	

Tab. A.9.: Selected assembly results for dataset R2.

R3	Unprocessed	CARE	CARE+RF	BFC	Musket
# contigs $\geq 50k$	3	51	63	57	52
# contigs	18,340	10,941	10,635	10,697	11,211
# misassembled	135	611	554	603	734
N50	7,859	14,196	14,749	14,673	13,742
NG50	5,506	10,251	10,624	10,470	9,886
	SGA	BCOOL	Lighter	Karect	
# contigs $\geq 50k$	43	24	48	53	
# contigs	11,298	13,872	11,056	10,820	
# misassembled	763	1,066	756	577	
N50	13,765	10,925	14,268	14,436	
NG50	9,806	7,727	10,162	10,263	

Tab. A.10.: Selected assembly results for dataset R3.