Parallelization of RazerS

Master's Thesis

Martin Riese

Freie Universität Berlin - Bioinformatik

15. August 2010

Erstprüfer:
Prof. Dr. Knut Reinert
Freie Universität Berlin
AG Algorithmische Bioinformatik

Zweitprüfer:
PD Dr. Jürgen Kleffe
Charité Berlin

Acknowledgment

I would like to express my sincere appreciation to all those who have contributed, directly or indirectly, to this Master's thesis.

First of all, I would like to thank supervisors Prof. Dr. Knut Reinert and PD Dr. Jürgen Kleffe for their guidance and for engaging with me in many fruitful discussions.

Furthermore, I thank the members of the working group Algorithmic Bioinformatics for their help and advice, especially David Weese and Konrad Rudolph.

I am very grateful to my entire family and my girlfriend Heather. Without their support this work would not have been possible.

Contents

Chapter 1

Introduction

1.1 Motivation

Since the first introduction of DNA sequencing by Maxam and Gilbert (1977) scientists developed different approaches to carry out this task. The term second generation sequencing combines methods that rely on massive parallel sequencing of short sequences. These short sequences are normally derived from larger ones using ultra sound or similar methods for fragmentation. Since the first introduction of second generation sequencing, (Shendure et al., 2005) the technology has made tremendous advances. While the early applications were rather slow and costly, today it is possible to sequence one billion base pairs per day (Roche Applied Science, 2010) at a cost of only $48.000 for one human genome (Illumina Inc., 2010). These improvements have made it possible to produce more and more data that subsequently needs to be processed. Furthermore, due to the competition of the three main companies (Roche Applied Science, Illumina and Applied Biosystems), this trend is likely to continue.

Although the companies use different approaches, the data produced by the experiments is quite similar. All of them generate short sequences, which we call reads. Their length varies from 36 to 600 base pairs. A single machine can produce up to 250 million reads a day (Roche Applied Science, 2010). Two approaches exist to reconstruct the complete sequence from these small fragments. First, a de novo assembly, arranges the reads based on their overlap. The second approach uses a reference sequence and maps the reads to it. The class of software that uses the latter technique is called read mappers, of which RazerS (Weese et al., 2009) is one example.

During the same time period in which second generation sequencing evolved, computer architecture underwent a significant change as well. Until the middle of the first decade of this century almost all personal computers (PCs) featured a single central processing unit (CPU) that processed calculations in a serial manner. But the growing energy consumption and the accompanying generation of heat from increasingly faster processors forced the chip designers towards an architecture with multiple processing units (cores), each providing only part of the computation power. Today most PCs feature CPUs with two or four cores. In the server market the number is even greater with 8 to 32 cores.

Most software that is written with the single core design in mind cannot make use of more than one processing unit (work in parallel) without adjusting algorithms and data structures. This is also true for the read mapper RazerS. Therefore, modifications in this direction are necessary to address the need for fast processing of growing sequence data. Other read mappers already support multi-core designs (see section 1.2), but they differ from RazerS in what kind of mapping they allow. This and an additional sensitivity control makes RazerS more valuable for some applications.

1.2 Related work

In the scientific community a wide variety of read mapping software is available, many of which feature a build-in multi-core option. Table 1.1 shows a selection. Two of them are described in detail below.

Table 1.1: List of read mapping software, their support of multi-core computation and the approach used for the implementation.

Read mapper	Multi-core	Remarks
BFAST (Homer et al., 2009)	Yes	By reads
Bowtie (Langmead et al., 2009)	Yes	By reads
BWA (Li and Durbin, 2010)	Yes	Not published
CloudBurst (Schatz, 2009)	Yes	MapReduce
GenomeMapper (Hagmann, 2009)	Yes	Not published
MAQ (Li et al., 2008)	No	-
MOM (Eaves and Gao, 2009)	Yes	Not published
Mosaik (Strömberg and Lee, 2009)	Yes	Not published
Mr. and Mrs. Fast (Alkan et al., 2009)	No	-
ProbeMatch (Kim et al., 2009)	No	-
RazerS (Weese et al., 2009)	Not yet	By reads (from version 3 on)
RMAP (Smith et al., 2008)	No	-
Segemehl (Hoffmann et al., 2009)	Yes	Not published
SeqMap (Jiang and Wong, 2008)	No	-
SHRiMP2 (Rumble et al., 2009)	Yes	By reads, using OpenMP
SOAPv2 (Li et al., 2009)	Yes	Not published
ZOOM (Lin et al., 2008)	No	-

1.2.1 Bowtie

Bowtie, (Langmead et al., 2009) like many other read mappers, uses the Burrows-Wheeler-Transformation to index the reference sequence. Based on that, the program processes one read after another, searching for matches in the index. In the parallel version, n threads consume reads from a synchronized queue, processing them individually and writing the results to a synchronized output file handler (Langmead, 2010).

1.2.2 CloudBurst

CloudBurst (Schatz, 2009) is a read mapping software that is especially designed to work in parallel. It is based on RMAP (Smith et al., 2008), which uses the q-gram lemma (lemma 2.1.3) to filter the reference sequence and then verify potential matches. To achieve parallelization CloudBurst utilizes Hadoop (http://hadoop.apache.org) an open source implementation of the MapReduce method developed by Google (Dean and Ghemawat, 2008).

MapReduce is designed to distribute workload over multiple processing units within one computer or over the network. It operates on interfaces that let the user plug in custom functions that are executed in three phases.

1 Map phase The first phase is the map phase (Map in this context means that the input is mapped to tasks and should not be confused with mapping of reads). In general, this step of the algorithm produces key-value-pairs that are then stored in a list.

With CloudBurst, the keys are the hash values of the q-grams (see section 2.1.3) at each position in the reference and at every q^{th} position in the reads. The value stores information about the origin of the q-gram, either the reads or the reference, as well as its position within the source.

2 Shuffle phase The second step of MapReduce sorts the list of key-value-pairs by the keys. For this particular application it groups together q-grams with the same sequence independent of their origin. Therefore, if the reference and a read share a q-gram with a certain hash value, they will be in the same section of the sorted list.

3 Reduce Phase In the last phase all elements from the list with the same key are passed on to a commutative function, which then calculates part of the results. In CloudBurst this function first separates the q-grams from the reads and the reference. It then generates the Cartesian product of these two sets and computes the alignment for each of its tuples. The result is a list with all matches that were triggered by q-grams with the same hash value.

1.3 Objectives

The main objective of this Master's thesis is the implementation of a parallel version of the read mapper RazerS. In detail this includes the following steps:

- Analyze the existing serial version of the program for data dependencies and other obstructions of concurrency.
- Design and examine different approaches to introduce parallelism.
- Implement at least one of them using the existing code and the C++ library SeqAn (Döring et al., 2008).

- Compare the performance of the parallel version with other read mappers that support multi-core computation.

RazerS uses algorithms and data structures from SeqAn. Therefore, a secondary objective is that all changes that are made to them also have to work outside of the context of RazerS.

1.4 Chapter summary

This thesis is subsequently divided into four chapters. The first provides the fundamentals needed for the understanding of the latter chapters. It begins with the definition of sequences, alignments, and q-grams as well as related data structures. Afterwards, we describe concepts in parallel hardware and software, which is followed by a brief sketch of sorting algorithms. The end focuses on the problem of read mapping.

In the following chapter we analyze RazerS regarding data dependencies and how they can be avoided in a parallel version. Building on that, the next section shows three approaches to introduce concurrency in the software, along with their advantages and disadvantages. Subsequently, we introduce the open addressing q-gram index. This is supplemented by some implementation details.

The results and discussion chapter first examines different approaches to solve the problem of waiting times in the parallelization by reads. The subsequent section compares the parallel version of RazerS with other read mappers that support multiple threads. In the end we have a closer look at the open addressing q-gram index and the performance of larger q-grams, which can be used due to this data structure.

The last chapter summarizes the results of the parallelization and the new q-gram index. It also lists remaining work and possible future improvements.

Chapter 2

Fundamentals

This chapter provides the fundamentals for the core of this thesis. It begins with some notations. Subsequently, we will introduce concepts of concurrent hardware and software as well as OpenMP, the application programming interface (API) that we used for the parallelization. This is followed by a short sketch of sort algorithms used in the software. Finally, it closes with an introduction of filtering algorithms, the description of the SWIFT algorithm and the way it is implemented in RazerS.

2.1 Notation

This section provides necessary definitions and notations of sequences, alignments, and q-grams.

2.1.1 Alphabets and strings

In this thesis we refer to a collection of values that are in a special order as a string or sequence. All of theses values are elements of a common set called an alphabet. More formally they are defined as:

Definition 1 (Alphabets and Strings)**.** An alphabet Σ is an ordered set of unique elements. A string T of length $|T| = n$ is a concatenation of n symbols (or characters) $T[i]$, where $0 \leq i < n$ and $T[i] \in \Sigma$.

The application of read mapping is based on genetic sequences. In this case the alphabet is defined as $\Sigma := \{a, c, g, t, n\}$, where the letters represent the different nucleotides or any nucleotide (n).

A continuous part of a string consisting of one or more characters is called a substring or infix. It is referred to in the following way:

Definition 2 (Substring)**.** $T[i..j]$ is a substring of T if $0 \leqslant i < j \leqslant |T|$. Where $T[j]$ is the first symbol following the end of the substring.

Two special types of substrings are prefixes, which always start at the first position of a string, and suffixes that end at the last position.

a) Hamming distance: $d_h = 3$

```
c t t g g c
|   |     |
c a t t g c
```

b) Edit distance: $d_e = 2$

```
c - t t g g c
|   | | |   |
c a t t g - c
```

Figure 2.1: Illustration of the Hamming and edit distance of the sequences "*attggc*" and "*acttgc*". The sequence of edit operations is C, I_a, C, C, C, D, C.

2.1.2 Distances and alignments

In sequence analysis the similarity between two sequences is often expressed through their distance, which can be defined in various ways. The most simple version is the Hamming distance. It is restricted to sequences of the same length and counts the number of conflicting characters.

Definition 3 (Hamming distance)**.** Let S and T be two strings of the same length n over the alphabet Σ. Then the Hamming distance is defined as:

$$d_h(S,T) = \sum_{i=0}^{n-1} \delta(S[i], T[i]) \tag{2.1}$$

where $\delta(s,t) = 1$, if $s \neq t$ and 0, otherwise.

A more complex measure is the weighted edit distance (Gusfield, 1997). It simulates the transformation from the first sequence to the second with four possible operations for this process: C - copy a character, S_c - substitute it with the character c, D - delete a character in the second sequence, and I_c - insert the character c in the second sequence. Each of them has a cost assigned to it. In most applications this is zero for copying and one for all other operations.

Definition 4 (Weighted edit distance)**.** Let S and T be two strings over the alphabet Σ. $E(S,T)$ is a sequence of edit operations over the alphabet $\varepsilon = \{C, S_c, D, I_c\}$ that describes a transformation from S to T, and E^* the set of all possible transformations.

$$d_e(S,T) = \min_{E \in E^*} \sum_{i=0}^{|E|} cost(E[i]) \tag{2.2}$$

In a biological context the edit distance is a better measure. It resembles DNA mutations, which include point mutations, single base insertions and deletions.

We speak of an alignment as the arrangement of two sequences. An alignment is called optimal if the distance between the sequences is minimal. In the case of the Hamming distance this is always the case but introducing additional insertions and deletions can change the edit distance.

Furthermore, alignments are grouped in three categories:

- Global: The whole sequences are aligned.
- Local: Substrings of two sequences are aligned.
- Semi-global: The substring of one sequence is aligned to a whole second sequence.

2.1.3 Q-grams

Substrings of a fixed length q are called (ungapped) q-grams. A string T of length n consists of $n - q + 1$ overlapping q-grams (from $T[0..q]$ to $T[n - q..n]$).

Definition 5 (Q-gram)**.** Let T be a string. Then $Q_i = T[i..i + q]$ is the q-gram at position i in T.

To represent them in a more compressed form, one often uses the hash value of q-grams.

Definition 6 (Hash value of a q-gram)**.** Let T be a string over the finite alphabet $\Sigma = \{a_1, ..., a_\sigma\}$, where σ is the size of the alphabet. Each of the letters in the alphabet has an assigned value $v(a_i) = i$. With Q as a q-gram in T its hash value is defined as:

$$hash(Q) = \sum_{j=0}^{q-1} v(Q[j]) \cdot \sigma^{q-1-j} \tag{2.3}$$

As an example we consider the q-gram $Q =$ "$atacg$" based on the alphabet $\Sigma = \{a, c, g, t\}$, with the values $v(a) = 0$, $v(c) = 1$, $v(g) = 2$, and $v(t) = 3$. Then $hash(Q) = 0 \cdot 256 + 3 \cdot 64 + 0 \cdot 16 + 1 \cdot 4 + 2 \cdot 1 = 198$.

Q-gram index A common method to store all q-grams of a string and make them readily available are q-gram indices. This data structure consists of two tables: positions and directory. The former holds all positions of a string (0 to $|T| - q$), at which a q-gram starts. It is sorted by the hash values of these q-grams. This means all positions of q-grams with a certain hash value are consecutive elements in this table.

The directory table has one entry for every possible q-gram hash value. Each entry holds a pointer to the first element in the positions table that stores a position of a q-gram with this value. If there is no such element, it points to the first one with a higher hash value or a virtual entry marking the end of the table.

Consequently, the range in the positions table that stores the positions for q-grams with a specific hash value h is $(directory[h], directory[h+1]-1)$. To make this consistent, the directory table has an additional element at the end for the maximal hash value plus one. Figure 2.2 gives a short example for a q-gram index.

String comparison Sequence analysis often uses the concept of shared q-grams to compare two or more strings with a certain similarity.

Definition 7 (Shared q-grams)**.** Let S and T be two strings. Then S and T share

$$\Big| \; \{ \; (i, j) \mid hash(S[i..i + q]) = hash(T[j..j + q]) \; \} \; \Big| \tag{2.4}$$

common q-gram, where $i \in [0,\ |S| - q]$, $j \in [0,\ |T| - q]$ and no value for i or j appears in more than one pair (i, j).

The theoretical foundation of the comparison of two sequences is the q-gram lemma.

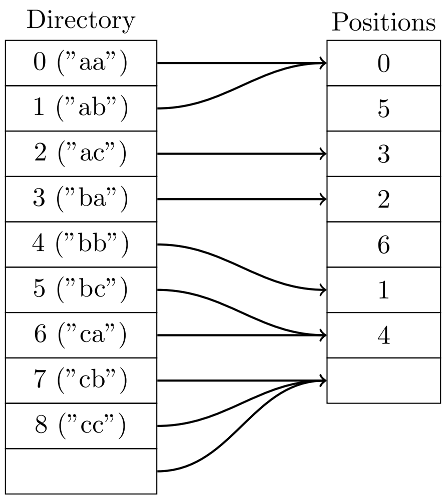


Figure 2.2: Example of a 2-gram index for the text $T =$ "*abbacaba*" over the alphabet $\Sigma = \{a, b, c\}$. To access all positions of 2-grams with a certain sequence, we first calculate its hash value h (e.g. $hash(\text{"}ba\text{"}) = 3$). Then we look up $directory[h]$ and $directory[h + 1]$, which point to the first entry in *positions* that contains a corresponding position and to the entry consecutive to the last one respectively. To provide a consistent behavior each table has an empty dummy element at its end, and entries $directory[h]$ for which no g-gram Q with $hash(Q) = h$ exist point to the first position of the entry in *positions* that stores a positions of a q-gram with hash value greater than h.

Lemma 1 (Q-gram Lemma)**.** Let T and S be two strings of length n with k differences. Then T and S share at least $w = n + 1 - (k + 1)q$ q-grams.

In the same way one can make assumptions about the similarity based on the number of shared q-grams.

Lemma 2 (Minimum Coverage)**.** Let T and S be two strings of length n with w shared ungapped q-grams. Then T and S match at least $w + q - 1$ positions. This value is called minimum coverage.

Gapped q-grams If the conflicting positions between two strings are spread out evenly over the whole length, they significantly reduce the number of shared q-grams and therefore the minimum coverage. This means that the expressiveness of this measures diminishes. The concept of gapped q-grams reduces this effect. It uses a shape to describe which positions in a q-gram are significant and which are ignored.

Definition 8 (Gapped q-gram)**.**

- A shape G is an ordered set of non-negative integers including 0 ($G = \{i_1, ..., i_q\}$, with $i_k < i_{k+1}$).
- A shape has two sizes: the cardinality $|G|$, which is the number of positions, and the span $s(G) = max(G) + 1$, the number of positions plus the number of gaps in between them.
- $G_j = \{j + i | i \in G\}$ is a positioned shape for $j \in \mathbb{N}$
- Let T be a string, then the gapped q-gram at position j is $\{T[g] \mid g \in G_j\} = T[G_j]$,

As an example, let $T =$ "*atcgtaga*" be a string and $G = \{0, 1, 3\}$ be a shape. Then the gapped q-gram at position 2 is $T[G_2] = T[2, 3, 5] =$ "*cga*". To visualize the effect on the minimal coverage, consider figure 2.3.

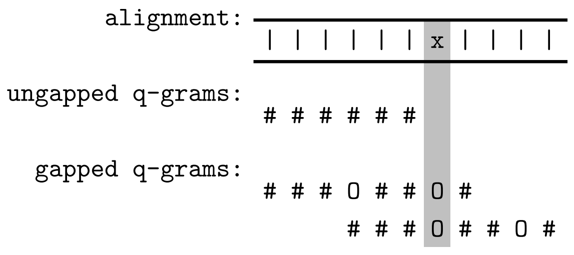


Figure 2.3: Comparison of ungapped and gapped q-grams. The alignments shows two strings that mismatch at one positions (position 6). Only one ungapped 6-grams can be placed over the matching positions. In contrast, two gapped q-grams of the same cardinality can cover nine of the 10 matching bases.

2.2 Parallelization

This section provides the fundamentals of concurrency. The first part gives an introduction to parallel hardware, followed by concepts used in parallel software. At the end we give a brief description of the API OpenMP (OpenMP API 3.0, 2008), which provides a framework for parallel programming.

2.2.1 Parallel hardware

Computers consist of three main parts: the memory that stores data and instructions, a processing unit - the CPU, and an instruction pointer that marks the current operation in a list (Sodan et al., 2009).

As stated in the motivation, recent computer designs allow parallel execution of code. This can be achieved through different means including multiple processor cores or hardware threads. The former approach copies all elements that are essential for a CPU, and integrates the duplicates into a common circuit. In contrast, hardware threads are more lightweight - only the instruction pointer is multiplied. If the instructions from one pointer cause the processor to wait for resources like the memory, it switches to the second pointer and continues to process the operations associated with it.

The data storage of a computer is organized in levels. The largest and typically slowest is the hard drive in comparison to the faster, but smaller main memory. Even faster access times are achieved with the cache memory. These hold copies of small parts of the main memory - normally between 32 kilobytes and 16 megabytes - and are positioned in close proximity to the CPU. If the processor has to read from the memory, it first looks in the lowest level cache. In the case that there is no duplicate, it checks the next higher level up in the memory. We call this behavior a cache miss.

In multi-core designs caches can be shared or private. Many architectures assign a private Level 1 cache (the smallest and fastest) to a single core and share all higher levels (see figure 2.4 for an example architecture). If two private caches hold a copy of the same variable and it is changed in one, the computer updates the value in the other.

2.2.2 Parallel software

In serial programs the arrangement of instructions follows a linear order. In contrast, parallel programs organize instructions in threads that work mainly independently from each other. The decision which thread performs a specific operation can be deterministic or may depend on the speed of the separate threads, which is influenced various factors.

However, threads usually are not completely autonomous. They use variables to exchange information that is important to coordinate their further work or share data to reduce redundancies. The latter is the case in many algorithms in bioinformatics that build upon large data structures. Problems occur if a program executes multiple writing operations on such a data structure in parallel, or if the correct behavior of a

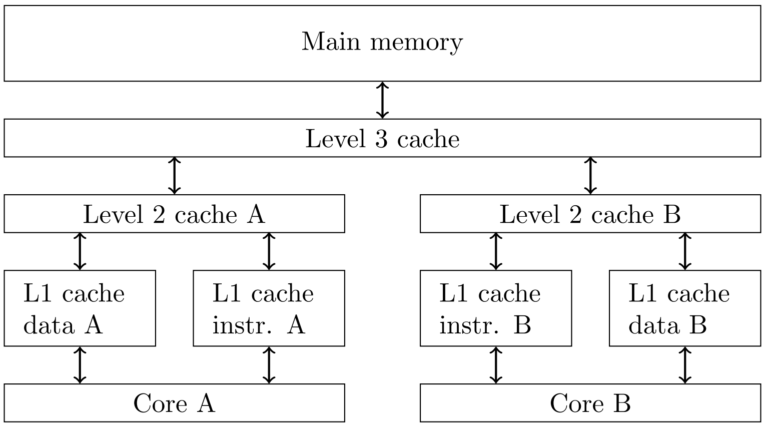


Figure 2.4: Example organization of computer memory in main memory and three levels of cache.

thread depends on the invariability of a value changed by another thread. Since the exact order in which threads access variables change with every run of the program and it is not clear which one is first, we speak of race conditions.

In the optimal case the design of a parallel program prevents it from entering such a situation. If that is not possible, spin locks (also called mutexes) can be used to protect selected variables. The idea is that a thread has to request the lock before it can execute a set of operations that depends on an exclusive use of a variable. If another thread already holds the lock, the first thread has to wait until it is released. Of course this means that the thread is idle as long as it cannot access the lock.

If the work that is done by one thread is split up and distributed further, we speak of nested parallelism.

A common measure to describe the improvements due to parallelization is the speedup. Kumar et al. (1994) define it in the following way:

Definition 9 (Speedup)**.** Let T_P be the physical runtime of the parallel program on P processor cores and T_S be the runtime of the fastest serial program solving the same problem. Then the speedup $S = T_S/T_P$.

Ideally, the speedup is equal to the number of cores. If not, the extra computational cost in the parallel program that is not present in the serial program is called overhead.

There are different sources of parallel overhead (Kumar et al., 1994), the first being inter core communication. If threads that run on different cores have to synchronize variables, they must use a level of the memory that is shared by the two cores, which can be comparable slow and therefore time consuming.

A second source are load imbalances. These occur if one thread has to wait for a another to update a variable in order to continue, or if a thread is done and a second is still working.

Extra computation is the last source of overhead. In some cases serial algorithms cannot be parallelized without some major modifications that introduce additional instructions. This includes the scheduling of threads.

2.2.3 OpenMP

There are different ways to introduce concurrency into computer programs, one of which is the API OpenMP. It provides a high level formalism for the programming languages C++ and Fortran, which is implemented by a number of compilers (table 2.1 lists a few of them).

Table 2.1: List of selected compilers that support OpenMP (OpenMP.org, 2010) and the latest supported version.

Vendor	Compiler	Supported version
GNU	gcc (4.4 and upwards)	3.0
GNU	gcc (4.2 and upwards)	2.5
Intel	C/C++/Fortran (10.1)	3.0
Microsoft	Visual studio 2008 C++	2.0

For C++, OpenMP defines a set of directives, clauses, and runtime library routines to model concurrency. In order to use them with GNU gcc, the compiler flag `-fopenmp` must be set and the program needs to include the header file `omp.h`. All directives start with `"#pragma omp"`, which is followed by the name, optional clauses and a line break. Their scope is the first following code block.

The parallel directive The `parallel` directive is the fundamental construct in OpenMP. It creates a set of threads (team) that executes the associated code block in parallel. The number of threads can be set using either the clause `num_threads(<num>)` or by calling the function `opm_set_num_threads(<num>)` prior to the directive.

If not restricted further, parallel execution means that the same code is run by each thread. In most cases this is not the desired behavior and nested directives are used to model the program flow. The simplest is the `single` directive, which runs the code with only one thread keeping the rest of the team waiting in the background.

The For directive The `for` directive requires that the first C++ statement past it to be a for loop that uses an index variable. It breaks up the whole range of the loop into chunks using the threads from the team of the the immediately surrounding `parallel` directive to process them in parallel. One of three strategies to distribute the chunks to the threads can be chosen using the `schedule(<stategy> [, <chunk size>])` clause:

- `static:` The chunks are assigned to the threads in a round-robin fashion.
- `dynamic:` Each thread requests a new chunk with completion of the prior until no chunks remain.

- `quided:` Similar to `dynamic` but the chunk size decreases over time.
- `auto:` The compiler decides which strategy to use.

The task directive To create a branch that is decoupled from the rest of the program flow, OpenMP offers the `task` directive. The code block associated to it is executed by one of the team members of the closest surrounding `parallel` directive. The directive`taskwait` merges the branches back together, waiting for all child tasks created within the current task. If there is not a task explicitly defined, the program itself is seen as the current task. The task directive is available in Open MP from version 3.0 on.

The critical directive To create and use spin locks, OpenMP provides two directives: `atomic` and `critical`. The former locks the first subsequent statement only. Additionally it requires the statement to update only one variable. On the other hand, `critical [(<name>)]` can lock a whole block. It also provides the possibility to specify a name. That way the lock can be used at different positions in the code.

Data sharing There are several clauses that can be used to specify data sharing strategies. `default(<data sharing clause>)` changes the default setting for all variables in a directive. As an argument it can take any of the following clause keywords or `none`. The latter enforces the strategy to be stated explicitly for each variable used in the block. All other clauses have the form `name(<list of variables>)`, of which two were used in the scope of this thesis.

- `shared:` The variables can be accessed by all parallel executed blocks.
- `firstprivate:` Each thread has a local copy of the variable, which is initialized with the value the variable had before the directive.

Even though the `shared` clause allows multiple threads to read and write on the same variable, they each have a local copy of the value, which has to be explicitly synchronized. `flush(<list of variables>)` updates the local or general copy based on which is older. To exchange a value between two threads, both need to call the directive. An implicit flush is called at the beginning of each block associated to a `parallel`, `for`, `task`, or `critical` directive.

2.3 Sorting algorithms

The algorithm described in the latter part of this thesis (see section 3.2.2) necessitates the ability to sort a list. There are many different algorithms for this task. In practice, quicksort (Hoare, 1962) is often the fastest option. This is why it is the default sorting algorithm of the C++ Standard Template Library (STL). One disadvantage of quicksort is that it relies on the >-operator only and ignores additional information that is available in some cases. Radix sort (Cormen, 2001, chap 8) uses intrinsic properties of the order between the elements to avoid this problem.

In the remainder of this section both algorithms are illustrated. A comparative evaluation for this particular application can be found in the results (see section 4.1.2).

Quicksort

Given a list, quicksort chooses one of the elements (randomly or deterministic), goes through the list, and divides it into two: one list containing all elements that are smaller than the chosen one and one containing the remainder. Subsequently, it does the same for the two new lists repeatedly until a list contains only one element. Algorithm 1 shows the pseudo code. The expected runtime of quicksort is $O(n \ \log \ n)$, where n is the length of the list.

Algorithm 1 QUICKSORT

```
$A$: list
$q$: begin position
$r$: end position

procedure QUICKSORT($A, p, r$)
  if $p < r$ then
    $q \leftarrow$ PARTITION ($A, p, r$)
    QUICKSORT($A, p, q - 1$)
    QUICKSORT($A, q, r$)
  end if

procedure PARTITION($A, p, r$)
  $x \leftarrow A[r]$
  $i \leftarrow p - 1$
  for $j \leftarrow p$ to $r - 1$ do
    if $A[j] \leq x$ then
      $i \leftarrow i + 1$
      EXCHANGE $A[i] \leftrightarrow A[j]$
    end if
  end for
  EXCHANGE $A[i + 1] \leftrightarrow A[r]$
  return $i + 1$
```

The parallelization of quicksort is rather simple. After the first partition step, the sub lists can be sorted independently and even the partitioning itself can be split up. One problem is that the partitions are normally of different lengths and therefore a balanced distribution of the calculations is difficult.

Radix sort

Radix sort assumes that the order of the elements can be broken down into partial orders. For instance, integer numbers can be sorted by single digits. This would lead not to a perfect order of the list, but to a partial one. Based on this assumption, the algorithm uses a stable sort and starts by sorting the list by the least significant partial order. It then repeats for all partial orders until it reaches the most significant one. Algorithm 2 shows the pseudo code for the case of integer numbers.

As the number of possible values per partial order is often rather small (e.g. ten for a digit in integer numbers), bucket sort is a good option for the stable sort

Algorithm 2 RadixSort

```
$A$: list
$d_{least}$: least significant digit to sort by
$d_{most}$: most significant digit to sort by

procedure RadixSort($A, d_{least},$)
  for $d \leftarrow d_{least}$ to $d_{most}$ do
    StableSort($A, d$)
  end for
```

algorithm. Bucket sort scans through the list twice. First, it counts the occurrences for all possible values. It then calculates the exclusive prefix sum over the counts. The prefix sums are start positions of the buckets. In the second scan, the algorithm moves the elements to the first free positions of their corresponding bucket to fill it.

The run time of bucket sort depends on the length of list n and the number of buckets b, although the latter only has a small effect. Worst, average and best run time are the same: $\Theta(n + b)$.

One way to parallelize bucket sort and therefore radix sort, is to divide the list into as many parts as there are threads. The algorithm then processes the parts independently with one intermediate synchronization step. Each thread initially counts the values in its part using a separate counter array. Subsequently, a modified prefix sum computes the start position for each bucket and thread. An outer loop iterates over the threads while the nested one goes over the possible values: $(t_1, v_1), (t_2, v_1), ..., (t_n, v_1), (t_1, v_2), (t_2, v_2), ..., (t_n, v_m), (t_n, v_m)$. Using this sum the threads can copy the elements from their part of the list to the new position without obstructing each other.

2.4 Read mapping

Read mapping describes the problem to semi-globally align reads to a reference sequence. The reads are usually between 36 and 600 base pairs long, where as the reference is normally a whole chromosome or even a genome (millions to billions of base pairs). A match at a specific position is defined as the best alignment within a close range. This notation is necessary as adding gaps close to the end of either sequence can produce many additional alignments that do not have any informative value. Depending on the actual problem, the alignments can be based on Hamming or edit distance.

Before the development of second generation sequencing, the BLAST algorithm (Altschul et al., 1997) was the preferred choice for this set of problems. But in the last couple of years the number of available read mappers increased significantly (see table 1.1). The main differences between the various tools regarding the feature set are the number of indels they allow for, the supported length of the reads, the sensitivity with which they find matches, and of course the speed.

A recent method in DNA sequencing are paired-end tags (Fullwood et al., 2009). It creates pairs of reads, which consist of a left and a right mate. The location of

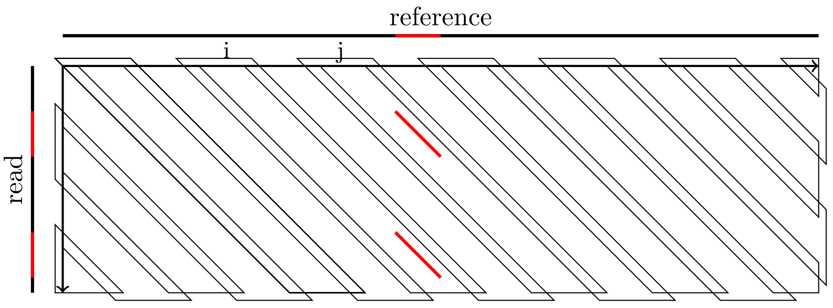


Figure 2.5: The SWIFT algorithms divides the virtual dynamic programming matrix between the read and reference sequence in over lapping bands. It counts shared q-grams (in red) between the two sequences for each of the bands.

these pairs is unknown as in general second generation sequencing, but the distance between the two mates ranges in a certain interval. For data from paired-end tag sequencing the problem of read mapping is extended by the constraint that matches for reads of a pair have to be in the proximity the experimental setup defines.

2.4.1 Filtering and the SWIFT algorithm

Calculating full semi-global alignments between all reads and the reference would take too long - even with fast algorithms like Myers' Bit-vector (Myers, 1999). For that reason most read mappers use an approach that filters the reference sequence first. They make use of the q-gram lemma (see lemma 1) or similar assumptions. The idea is to find regions in the reference that share enough q-grams with a read that there is a high probability for a match and to exclude parts that are not similar enough from a further examination.

The SWIFT algorithm by Rasmussen et al. (2006) can be used to apply this idea of filtering. Originally developed for local alignments of two long sequences it can be modified to operate with semi-global alignments. As illustrated in figure 2.5 the algorithm divides the virtual dynamic programming matrix between read and reference sequence into overlapping bands. A shared q-gram appears as a short diagonal stretch (red) in the matrix. If the number of q-grams per band exceeds a certain threshold, a hit is reported. These hits, which are tuples of read and substring of the reference, are then subject to further verification using an arbitrary alignment algorithm.

To count the shared q-grams the algorithm first builds a q-gram index over all reads. It then uses a sliding window approach to go over the reference sequence. For each q-gram it calculates the hash value and looks up all occurrences in the q-gram index. Depending on the relative position in the read the counter of the corresponding band is incremented. In figure 2.5 for example the red q-gram is present twice in the read. As a consequence, the algorithm increments the counters of both bands (i and j) by one.

Once a counter exceeds a certain threshold, SWIFT creates a hit consisting of a substring of the reference sequence and the read ID corresponding to the band whose counter was updated. As boundaries of the reference substring it uses the outer extend of the band in the direction of the reference sequence (see figure 2.6).

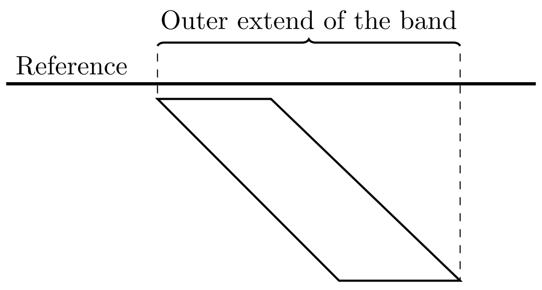


Figure 2.6: Outer extend of a band. This interval is used as the boundaries of the reference substring in a reported hit.

The overlap guarantees that a valid match appears in at least one band. If the number of insertions and deletions in a match is not maximal, it can happen that a match lies completely in an overlap. Therefore, the counters of both overlapping bands would reach the threshold and trigger a hit. This results in duplicate matches and one on of them can be removed without loosing information.

Another detail is the reuse of counters. Because of the large difference in length between the two sequences, each band is active for a small part of the algorithm only. Once the sliding window is past the last position of a band, it would never update its counter again. Instead, SWIFT recycles the counters. It resets the value of the counter to zero once the band is out of reach and uses it for the band that comes into the scope of the window next.

2.4.2 SeqAn and RazerS

SeqAn

SeqAn (Döring et al., 2008) is an open source C++ library focusing on sequence analysis. It was developed to make efficient implementation of common data structures and algorithms available to other developers.

For search and some alignment algorithms, SeqAn provides a finder-pattern-interface to hide the implementation from the users. This interface consists of three main parts:

- The finder holds information about the sequence that is searched, normally the longer one. It also stores possible results produced by the underlying algorithm.
- The pattern describes one or more strings that are searched. This can be a pointer to the sequence in the easiest case, or data structures that are created by a preprocessing step.
- The find function takes one finder and one pattern as arguments. Upon its call it calculates the next occurrence or alignment of the pattern and returns `true`. If it cannot find more, the return value is `false`. Iterative calls of the function are used to retrieve all occurrences or alignments.

The implementation of the SWIFT algorithm is accessed through this interface as well. The SWIFT finder has three main member variables: an iterator on the reference sequence that is used to produce the q-grams, an array of repeat regions that are skipped by the sliding window, and a list of hits. Each of the hits consists of a read ID and a substring of the reference (in this case reduced to its begin and end position). The pattern holds the q-gram index over the reads and parameters of the bands. This includes their position, counters for shared q-grams, and the thresholds for triggering a hit.

Myers' Bit-vector algorithm, named after its author (Myers, 1999), is a fast alignment algorithm, which uses bit vectors (VN and VP) to represent the dynamic programming matrix. In the first step it preprocesses one of the aligned sequences and calculates masks representing this string. The pattern stores both, the bit vectors and the masks. For this algorithm a simple finder is used. It consists of an iterator on the second sequence only. As result Myers' Bit-vector algorithm does not produce a full alignment but calculates the number of mismatching positions in it and the end position of the alignment.

An alternative to Myers' is the Hamming pattern. It only holds an iterator on the second sequence, in our case the read. The corresponding find function requires a simple finder as the second argument and compares the two strings allowing mismatches only. Figure 2.7 shows the UML diagrams of the data structures explained above.

a)

	SWIFT finder
+	curPos: Iterator
	(on reference sequence)
+	repeats: [Pair<int, int>]
+	hits: [SwiftHit]

b)

	SWIFT pattern
+	index: QGramIndex
+	counters: [int]
+	thresholds: [int]
	(for triggering hits)

c)

	Simple finder
+	curPos: Iterator
	(on reference sequence)

d)

	Myers' pattern
+	k: int
	(allowed mismatches)
+	bitMasks: [unsigned long]
+	VN, VP: unsigned long
+	score: int

Figure 2.7: UML diagram of finder and pattern data structures. a-b) SWIFT, c-d) Myers'.

RazerS

RazerS by Weese et al. (2009) uses SeqAn to combine the SWIFT and Myers' Bit-vector algorithm to a read mapping software. It operates in three phases: (1) preprocessing, (2) filtering and verification, and (3) formatting and writing the results to a file. To simplify matters, we group the following aspects as filter: the SWIFT finder, SWIFT pattern and find function. The verifier is composed of: the simple finder and Myers' or Hamming pattern.

The program starts with parsing the command line options and fetching the reference and reads sequences from the files. Subsequently, it builds the q-gram index over the read sequences. If the user selects the option to search for matches with indels, RazerS generates a list of Myers' patterns, one for each read, which are stored in the verifier. Otherwise it uses Hamming patterns that are simpler and therefore can be created on demand.

After the preprocessing is done, the software enters the filter and verification phase (see figure 2.8). The filter iterates over the reference sequences and generates hits as described in section 2.4.1. Once a hit is created, the program stops filtering and passes it on to the verifier, which calculates a semi-global alignment and compares the score with a threshold. If the score is high enough, the alignment is called a match between reference and read. The software then creates a match object (see figure 2.9, b) and appends it to a list called `alignedReadStore`.

These two steps are repeated until the filter reaches the end of the reference sequence. If more than one reference sequence is given, RazerS repeats the filter and verification phase for each of them. To compute matches on the reverse complement of the reference sequence, the program simply turns it around and processes it again.

If the size of the `alignedReadStore` exceeds a certain size during the second phase, RazerS calls two functions. `maskDuplicates` sorts the matches by their read ID and the end position of the infix, goes through the table, and removes all duplicates. The reason for why these can exist is the overlap of two bands.

`compactMatches` reduces the size further using the parameter m, which specifies the maximal number of matches RazerS reports per read. Its default value is 100. The function sorts the matches again; primarily by their read ID and secondarily by their score. It then iterates over the list and throws out all matches other than the m best of each read.

Additionally, RazerS performs one of two actions for reads with more than m matches depending on which option chosen by the user. If `--purge-ambiguous` is selected, it disables the corresponding read completely so it does not trigger any more hits and removes all existing matches of this read.

Without `--purge-ambiguous`, the program first calculates the value k_{worst}, which is the number of mismatching positions in the worst match of the read. Based on $k_{\text{worst}} + 1$ the program then uses the q-gram lemma (see lemma 1) to calculate a new threshold for reporting hits and writes it to the SWIFT pattern. This new threshold ensures that newly triggered hits lead the better matches than the worst existing one, and therefore reduces the number of verifications.

In the third and last phase, RazerS sorts the matches again. It computes full alignments if requested, and writes the results to a file in a format specified in the options.

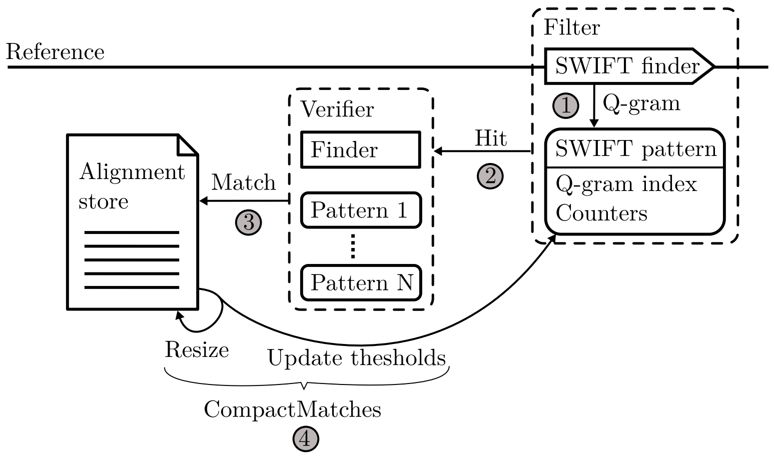


Figure 2.8: Filter and verification phase in RazerS. (1) The SWIFT finder iterates over the reference sequence and produces the hash value for the q-gram, which then is looked up in the q-gram index in the SWIFT pattern. The program increments the counters accordingly and if one reaches a certain threshold, a hit is reported (2). Depending on the command line arguments RazerS uses a preprocessed Myers' pattern or an newly created Hamming pattern to verify the hit. (3) If the alignment between read and reference substring is good enough, it is called a match and stored in the `alignedReadStore`. (4) If the size of the `alignedReadStore` exceeds a certain level, the program calls the `compactMatches` function, which removes duplicate matches and updates the thresholds in the SWIFT pattern depending on the number of matches per read.

a)

	Hit
+	infix: Infix
+	readID: int

b)

	Match
+	infix: Infix
+	readID: int
+	score: int

Figure 2.9: UML diagram for the a) Hit and b) Match data structures. The filter passes the former one to the verifier. If it is confirmed, the verifier adds the score to create a Match object, which it stores in the `alignedReadStore`.

Paired-end tag version

To process reads from paired-end tag experiments a modified version of RazerS is used. Instead of one index over all reads it creates one over the right mates and another one over the left mates. Furthermore, the filter and verification phase is split up as well. The right reads are processed as in the standard version of RazerS.

However, the algorithm for the left side is different. The program processes the left mates only if there is a match on the right side. To do so it introduces two new variables: L_{pos}, which stores the position up to where the reference sequence was filtered with the left mates. L_{queue} hold hits of left mates that were found so far.

If RazerS encounters a match for the read R on the right side, it stops and switches to the left. The algorithm then goes through the following five steps and returns to the right side afterwards.

1. Calculate the range $(L_{\text{begin}}, L_{\text{end}})$, in which a valid match of the mate of R can lie.
2. Go through L_{queue} and throw out all hits that are left of L_{begin}.
3. Filter the reference sequence from L_{pos} to L_{end} with the left index and save all new hits in L_{queue}. Then set $L_{\text{pos}} = L_{\text{end}}$.
4. Go through L_{queue} and verify all hits that belong to the mate of R.
5. If there are matches, take the one with the best score and store it in `aligned-ReadStore` together with the match of R.

Chapter 3

Analysis and methods

This chapter begins with the analysis of RazerS in regard of data dependencies that can cause problems in a parallel version, and shows strategies to avoid them. Subsequently, we describe three approaches to introduce concurrency in RazerS as well as their advantages and disadvantages. This is followed by a section about the open addressing q-gram index. In the last part we will provide some implementation details.

3.1 Data dependencies in RazerS

Some of the data structures like the sequences and the q-gram index are read only after the preprocessing phase. Other variables including the Myers' pattern, `alignedReadStore` and SWIFT pattern are updated during the run of the program. If multiple threads operate on the latter category in parallel, race conditions occur. The remainder of this section studies data dependencies in RazerS and shows possibilities to avoid them.

3.1.1 Myers' pattern

The Myers' patterns, used during the verification of hits, contains bit masks for each read (see paragraph 2.4.2). Since normally multiple hits occur per read, it is beneficial to preprocess and reuse the masks. Additional to these constant data structures, Myers' patterns also contain variables that change during the execution of the algorithm. If two threads share the list of patterns and verify the same read in parallel, they cause race conditions.

Different strategies can be used to avoid this behavior. Probably the most straightforward solution is to copy the whole list of Myers' patterns assigning one duplicate to each thread. For one million reads of length 100 and 8 threads this would add up to additional $7 \cdot 10^6 \cdot 124\ \textit{byte} = 868$ MB. Even though modern servers feature quite big memories this extra space requirement should be avoided, especially under the assumption that the number of available cores and therefore of threads can be higher than eight and is likely to grow even more in the future.

A second solution is to introduce spin locks for the patterns allowing only one thread to use them at a time. As a result, a second thread that is requesting the data structure would remain idle causing waiting times. Each lock can either protect a single pattern, or several at once. The former approach requires more variables to realize the locks, whereas the latter leads to more frequent blocking of other threads. Furthermore, this method requires communication between the threads, which results in additional computation.

Another approach divides the reads and therefore the Myers' patterns into subsets, each used by only one thread. This partitioning can be static throughout the execution of the software or change dynamically. To realize the latter, the program collects a greater number of hits, sorts them by the read IDs, and then divides the list so that no two sublists contain a hit with the same read ID. Algorithm 3 shows the pseudo code.

Algorithm 3 PartitionHits

```
$H$: list of hits
$n$: number of parts

procedure PartitionHits($H, n$)
  SortByReadId($H$)
  $partitions \leftarrow [\ ]$
  $partSize \leftarrow$ Length($H$)$/n$
  $begin \leftarrow 0$
  for $i \leftarrow 1$ to $n$ do
    $pre \leftarrow H[i \cdot partSize - 1]$
    for $pos \leftarrow (i \cdot partSize)$ to Length($H$) do
      if $pre \neq H[pos]$ then
        Break
      end if
    end for
    $tuple \leftarrow (begin, pos - 1)$
    Append($partitions, tuple$)
    $begin \leftarrow pos$
  end for
  return $partitions$
```

Finally, one can change the finder-pattern-interface and remove the changing variables from the pattern and store them separately. The bit masks and other constants remain at their original location. The `find` function cannot host the variables, because it is called multiple times to complete the algorithm and with every call the values are reset. As an alternative, we introduce a new data structure `_PatternState` (see figure 3.1) and change the notation of the `find` function to accept this additional argument.

The last approach is likely the best since it also reduces the space needed for the array of Myers' patterns due to the outsourcing of variables, which now exist only once per thread. However, it requires a modification of the find interface in SeqAn.

a)

	Myers' Pattern
+	k: int
+	bitMasks: [unsigned long]

b)

	Myers' _PatternState
+	VN, VP: unsigned long
+	score: int

Figure 3.1: UML diagram for the a) reduced Myers' pattern b) and the corresponding pattern state.

3.1.2 Appending `alignedReadStore` and `compactMatches`

After successfully verifying a hit, RazerS appends a new match to the `alignedRead-Store`. This write operation can cause conflicts, if two or more threads attempt to execute it at the same time. There are two solutions to this problem.

The first approach uses spin locks. If a thread has to append the `alignedRead-Store`, it requests the lock. If it succeeds, the thread performs the operation and releases the lock afterwards. Otherwise, it waits until the lock is released by the thread, which is in its possession. This behavior leads to waiting time and requires data exchange between caches, which slows down the program.

Another solution is to split up the `alignedReadStore` and provide a copy to each thread. In this scenario each thread can execute the append operation without restraining other threads. It requires no communication and causes no waiting times. To integrate the multiplied store in the existing program flow they can either be combined to one main store at the end of the filter and verification phase or the code of the third phase can be modified to deal with more than one store.

If the `alignedReadStore` exceeds a certain size, RazerS calls the functions `mask-Duplicates` and `compactMatches`, which reduce the size of the table. We can use the mechanism from the append operation described above to prevent conflicts in this case as well. However, `compactMatches` also updates the thresholds in the SWIFT pattern. It relies on counting the matches per read that are in the `alignedReadStore`. If the data structure is split up, the matches are distributed with it and the `compactMatches` function miscounts. Therefore, the thresholds cannot be adjusted accordingly and more hits than necessary are reported and verified.

Furthermore, in the case that `--purge-ambiguous` is selected, this would lead to the situation where the program disables the corresponding read globally but removes the existing matches from one `alignedReadStore` only, whereas they remain in a different one and are falsely part of the output.

As solution we propose a combination of the two approaches above. Each thread that appends the `alignedReadStore` holds a local list to save the matches temporarily. Additionally, the program has one main store, which is protected by a spin lock. In regular intervals, a thread requests the lock to transfer its matches to the main store. If it can require the lock, the thread performs the append operation and if necessary, calls `maskDuplicates` and `compactMatches`. Otherwise, it continues working and tries again after the next interval. This method requires communication between the threads but it can be minimized by choosing appropriate interval lengths.

For this approach to work with `--purge-ambiguous` the program keeps a list of read IDs for which matches were removed from the main store. If a second thread adds matches to it, it first compares the read IDs with the ones in the list and removes matches if necessary. A read Id can be removed from the list once each of the threads has appended to the main store after the ID was added.

An alternative to this last approach uses separate `alignedReadStores` under the assumption that hits from a specific read are always verified by the same thread. This way, all matches of this read are stored in the same copy of the table and `compactMatches` counts correctly.

3.1.3 Counters in the SWIFT pattern

The SWIFT pattern holds counters, which RazerS uses to keep track of the shared q-grams between reference and reads. The number of these counters (n_{count}) is linearly proportional to the number of reads (n_{reads}) and reference sequences (n_{ref}) that are filtered in parallel.

$$n_{\text{count}} = n_{\text{reads}} \cdot n_{\text{ref}} \cdot \alpha \tag{3.1}$$

Where α depends on the length of the reads and the number of allowed mismatches.

This means that if two threads filter two reference sequences with all reads at the same time, the program needs twice as many counters as the serial version.

3.1.4 Sequences

RazerS computes the reverse complement of the reference sequence in place to map the reads in reverse direction. This means that the program modifies the original sequence. Therefore, the only way to compute the forward and reverse matches concurrently is to create a copy of the reference sequence.

After the preprocessing, RazerS accesses the read sequences through the q-gram index and the Myers' pattern only, or in a read only manner, if the Hamming distance is chosen. Since the index is read only as well, the only data dependency over the read sequences is through the Myers' patterns, which is described above (see section 3.1.1).

3.2 Parallelization of RazerS

RazerS shows several areas where parallelizations are possible. Two of them are the construction of the q-gram index and the calculation of the reverse complement of the reference sequence. However, the most time consuming and therefore the most beneficial is the filter and verifications phase. In the remainder of this section we describe three approaches to introduce concurrency in this part of the software. For all of them we assume that the best number of threads is equal to the number of available processor cores.

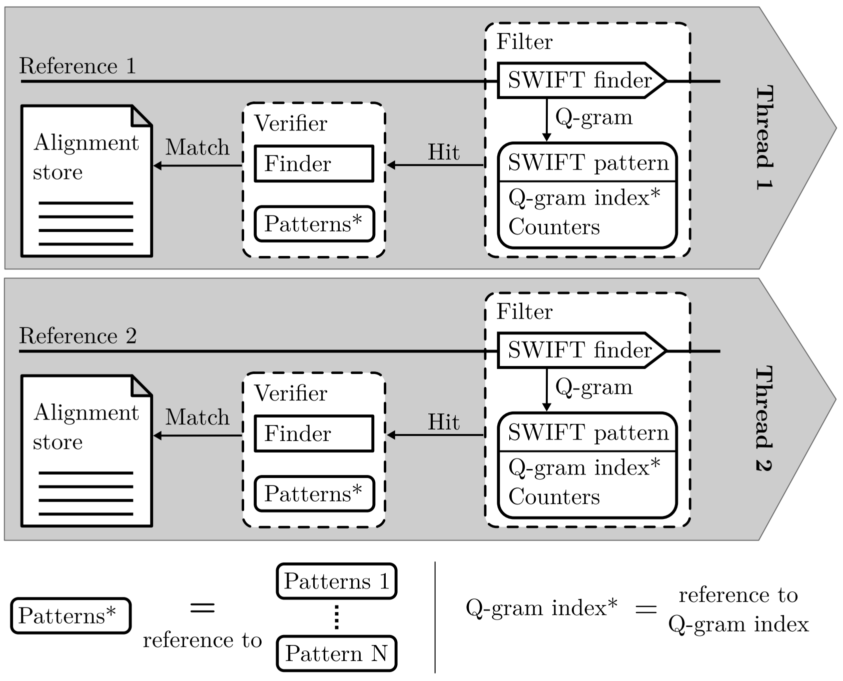


Figure 3.2: Parallelization of RazerS by reference sequences. Each thread holds its own filter and verifier, that work on different parts of the reference but filter and verify for all reads. To prevent duplication, the threads hold pointers to the shared q-gram index and list of Myers' patterns instead of the data structures themselves.

3.2.1 Parallelization by reference

The parallelization by reference is the first approach we are going to examine. The idea is to iterate over the reference sequence with one filter per thread instead of just a single one. For each filter the program also initializes a separate verifier that is coupled to it, and processes all hits that are produced by its associate. Figure 3.2 shows an illustration of this approach.

With the multiplication of the filter, copies of the SWIFT pattern are created as well. Each instance holds its own set of counters for the shared q-grams. The q-gram index can be shared by all threads. Since the filters search hits for all reads, the verifier must have access to all Myers' patterns and we need to apply one of the exit strategies described in section 3.1.1. However, the biggest disadvantage of the parallelization by reference sequence is appending the `alignedReadStores` and keeping track of the matches. Each of the solutions shown in section 3.1.2 requires synchronization between the threads.

Another important aspect of parallelization by reference is the partitioning of the sequences. In virtually all use cases, the number of given sequences will not match the optimal number of threads. Furthermore, we can expect the lengths to vary by a large degree. The ratio between the length of the biggest and smallest human chromosome for example is five to one. Because of these characteristics of the input data, an artificial partitioning of the sequences is necessary. If the segment boundaries do not lie at either end of a low complexity repeat region, which is skipped by filters, the single segments have to overlap. This overlap needs to fit the longest of the reads in order to find all matches.

Because of repeats and other bias in the sequence, segments of the same length will take different times to process. An advantage of parallelization by reference is the easy redistribution of work. If one thread is done with its segment, the program can split the remaining sequence of another thread in half and reassign a part of it. Since each thread holds all data structures necessary to filter and verify for all reads only the SWIFT finder has be to be modified, which is not a problem due to its small size.

Extending the parallelization by reference sequence to the case of paired-end tags is strait forward. All data structures that exist multiple times in the standard version are duplicated for the left and right mates as well. Additionally, the overlap for an artificial partitioning of the reference sequence has to be larger to fit both mates of a pair and the required gap between them.

3.2.2 Parallelization by reads

A second approach of introducing concurrency is by reads. Like for parallelization by reference, each thread holds a pair of filter and verifier. The difference is that because of the separation by reads, the two large data structures q-gram index and list of Myers' patterns can be split up as well (see figure 3.3). This holds the following advantages.

Since each filter only needs to look up the q-grams found in a subset of the reads, the index has to be over this subset only. This means the program constructs as many separate q-gram indices as there are threads, which can be done in parallel. Especially in the case of short reference sequences and large sets of reads, this improves the overall run time. Since the individual subsets of reads can be rather small for high numbers of threads, we use open addressing q-gram indices, which promise to reduce the required space compared to traditional indices (see section 3.3 and 4.3).

The second advantage is that because each read is processed by one of the threads only, the data dependency over the Myers' patterns is resolved without changing the code. The same is true for the `alignedReadStore` and `compactMatches`. If each thread has its own `alignedReadStore`, all matches of one read are saved in the same location and the `compactMatches` counts correctly.

Like the parallelization be reference sequence the approach by reads as well can be used in the case of paired-end tag reads.

The major problem that arises with the parallelization by reads is the difference in runtimes of threads. Due to the experimental setup of second generation sequences, reads are normally well distributed. Nevertheless, some reads match more often than others, especially if they originate from repeat regions. This means extra computation for the thread that processes them.

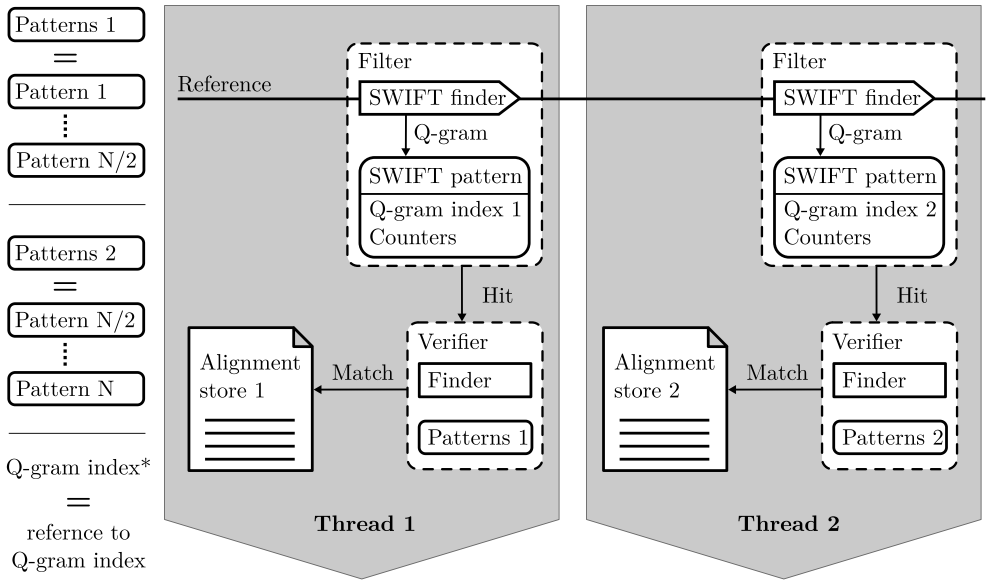


Figure 3.3: Parallelization of RazerS by reads. Each thread holds a pair of filter and verifier that work with a distinct subset of reads, but filter the same reference sequence. Because the reads are distributed between the threads, each SWIFT pattern holds a q-gram index over its subset and only part of the Myers' patterns.

Because q-gram indices encode which reads a filter can search for, this part of the algorithm cannot be changed, once the data structure is created. As a consequence it is not possible to redistribute part of the filtering to a finished thread. Instead, we are going to examine two different approaches.

Multiple subsets per thread

The first method separates the reads into more subsets than there are threads available. During the filter and verification phase the algorithm goes over the reference sequence in windows. The program assigns one subset to each thread, which then filters the window and verifies the created hits. A thread that is done with its current task continues with one of the remaining subsets until none is left. If that is the case, the thread waits until the other threads finish as well. The program repeats this cycle for the next window until it reaches the end of the reference sequence.

Because the reference sequence is used repeatedly for each subset of reads, we keep the window at a size that fits into the cache to counter cache misses. We expect a reduction of the waiting time because a thread that is slower will process less of the subsets, while faster ones use their excess time to process more. Section 4.1.1 shows that this is true for some parameters. However, with an increasing number of subsets overhead computation is necessary, which overcompensates this effect.

Distribution of the verification

As mentioned earlier in this section, a redistribution of the filter step is not feasible since it would require to rebuild the q-gram indices for smaller subsets or to separate the positions found in the index by read IDs. Either approach is time consuming and therefore would not lead to improvements.

Contrarily, the verification can be distributed amongst several threads. To do so the program has to collect a larger number of hits before it starts the verification. However, problems arise through the data dependencies over the Myers' patterns, `alignedReadStore`, and function `compactMatches` (see section 3.1).

To avoid the former, we have three different options: Spin locks, partition the hits by their read IDs, or rewrite the Myers' pattern. As sorting and partitioning takes only about 2 to 4% of the time needed to verify the hits afterwards (see section 4.1.2), it is a fast and minimally invasive option. An alternative is the `_PatternState` mentioned in section 3.1.1.

In either case, a thread that finishes early has to communicate this to the other threads allowing them to use its resources. We introduce the "array threads per subset" (*tps*) that stores the number of thread that can be used to verify hits from one subset of reads. For each subset it has one entry, which is initialized with one. Once a thread is done, it increments the first entry to its right that is not zero by the number it was allowed to use. A thread that is still working that has an entry greater than one redistributes part of the verifications to the threads left of itself. Algorithm 4 shows the pseudo code.

The data dependency over the `alignedReadStore` is resolved by introduction of a duplicate for each thread, which causes the program to write matches of the same read to different tables and therefore `compactMatches` to miscount. However, because the redistribution takes place for a short part of the whole program run only, the majority of matches of a single read are still stored in one of the stores. As consequence we can ignore the remainder.

To be able to remove all existing matches in the case that `--purge-ambiguous` is selected, we reset the threshold in the SWIFT pattern but leave matches in the `alignedReadStores` until they are combined at the end of the second phase. A last call of `compactMatches` on the main store before the output removes them.

3.2.3 Pipeline

The previous two methods work with pairs of filter and verifier - one per thread, and only one of the members of each pair is working at a time. In contrast, the pipeline approach uses specialized threads that hold either a filter or a verifier. In the simplest version one thread filters the sequence and produces hits, whereas a second one consumes them by verification. To buffer variations in the rates at which hits are produced and consumed a queue holds all hits that are not verified yet (see figure 3.4).

Most computers feature more than two cores, therefore a design with one filtering and one verifying thread does not make full use of the hardware. A more complex

Algorithm 4 ProcessReferenceSeq

```
$tps$: array with number of threads that verify hits from each subset of reads
$seq_{\text{ref}}$: reference sequence
$thread[\,]$: threads
$n_{\text{threads}}$: number of threads $id_{\text{thread}}$: ID of this thread

procedure ProcessReferenceSeq($seq_{\text{ref}}$)
  repeat
    $hits \leftarrow$ FilterWindow($seq_{\text{ref}}$)
    if $tps[id_{\text{thread}}] = 1$ then
      VerifyHits ($hits$)
    else
      $parts \leftarrow$ PartitionHits($hits, tps[id_{\text{thread}}]$)
      for $id_{\text{rel}} \leftarrow 1$ to $tps[id_{\text{thread}}]$ do
        $id_{\text{abs}} \leftarrow (n_{\text{threads}} + id_{\text{thread}} - id_{\text{rel}})$ Mod $n_{\text{threads}}$
        VerifyHitsWithThread ($parts[id_{\text{rel}}], thread[id_{\text{abs}}]$)
      end for
    end if
  until End of reference sequence is reached
  UpdateTps($tps[\,], id_{\text{thread}}$)

procedure UpdateTps($tps[\,], id_{\text{thread}}$)
  $step \leftarrow 0$
  $id_{\text{abs}} \leftarrow 0$
  repeat
    $step \leftarrow step + 1$
    $id_{\text{abs}} \leftarrow (id_{\text{thread}} + step)$ Mod $n_{\text{threads}}$
  until $tps[id_{\text{abs}}] > 0$
  $tps[id_{\text{abs}}] \leftarrow tps[id_{\text{abs}}] + tps[id_{\text{thread}}]$
  $tps[id_{\text{thread}}] \leftarrow 0$
```

architecture can solve this problem. Different combinations of filters, queues, and verifier are possible.

If the program instantiates multiple filters, they can work on different reference sequences, separate subsets of reads, or both. In either of the cases the queue that collects the hits can be private or shared by all or groups of filters. Depending on these designs, the same restrictions as for the parallelization by reference or reads apply.

Let R_{in} be the set of reads for which filters push hits in a queue. A verifier consuming from this queue can work with hits of all reads in R_{in}, or multiple verifier can distribute them amongst each other. A problem with the latter case is that hits for two or more subsets of reads are not evenly distributed over the reference. This can lead to a situation, in which one consuming verifier runs out of hits to process, while another one cannot keep up. This consequently, results in an increased size of the queue as well as in waiting times for the thread holding the first verifier. On the other hand, this partitioning avoids the data dependency over the Myers' patterns.

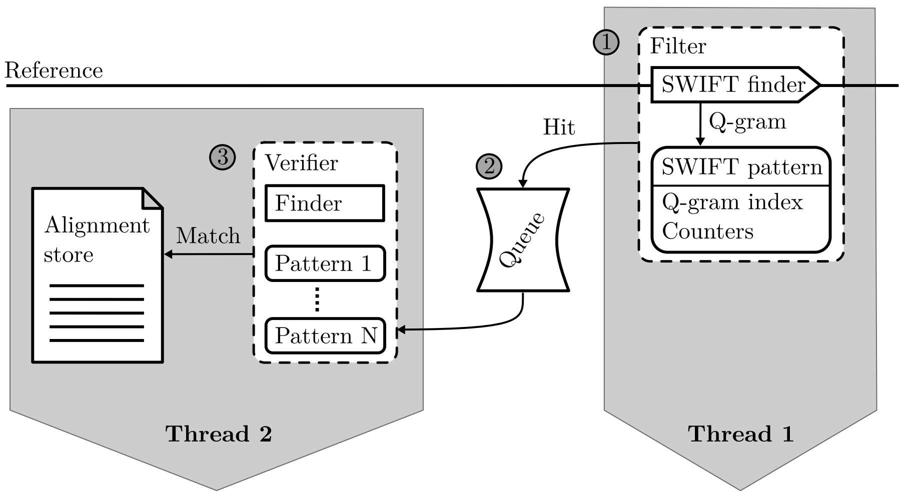


Figure 3.4: Parallelization of RazerS using the pipeline approach. 1) One thread filters the reference sequence and produces hits. 2) A queue temporarily stores the hits. 3) A second thread consumes the hits, verifies them, and stores resulting matches in the `alignedReadStore`.

The main disadvantage of the pipeline approach is the ratio between the time needed to filter and to verify hits. This proportion heavily depends on the data and the parameters of RazerS. This includes the option to allow insertions and deletions. Furthermore, within one run of the program the ratio can change by factor two or more (see figure 3.5). Because of this variation, either the filter or the verification threads will wait while their counterparts still work. A dynamic adjustment is possible but requires additional logic and communication between the threads, which will slow the software down.

A second problem is that the pipeline approach cannot be extended to paired-end tag reads, because of the dependency between left and right mates.

3.3 Open addressing q-gram index

In section 2.1.3 the q-gram index was introduced. It uses the hash value of a q-gram directly to address the corresponding entry in the directory table. As a consequence the table needs one entry for each possible q-gram hash value and the size of the index is in direct correlation to q, the alphabet size σ and the length of the indexed sequence n:

$$size_{index} = size_{buckets} + size_{positions} = (\sigma^q) + (n - q + 1) \tag{3.2}$$

For indices over a set of m short strings (like reads) the formula for $size_{\text{positions}}$ changes slightly to $m(n - q + 1)$. If each number in the index is encoded in 4 byte, the index over a DNA sequence of length 10^6 and $q = 14$ would be of size:

$$(4^{14} + 10^6 - 13) \cdot 4 = (268,435,456 + 999,987) \cdot 4 \; byte \approx 1 \; Gb \tag{3.3}$$

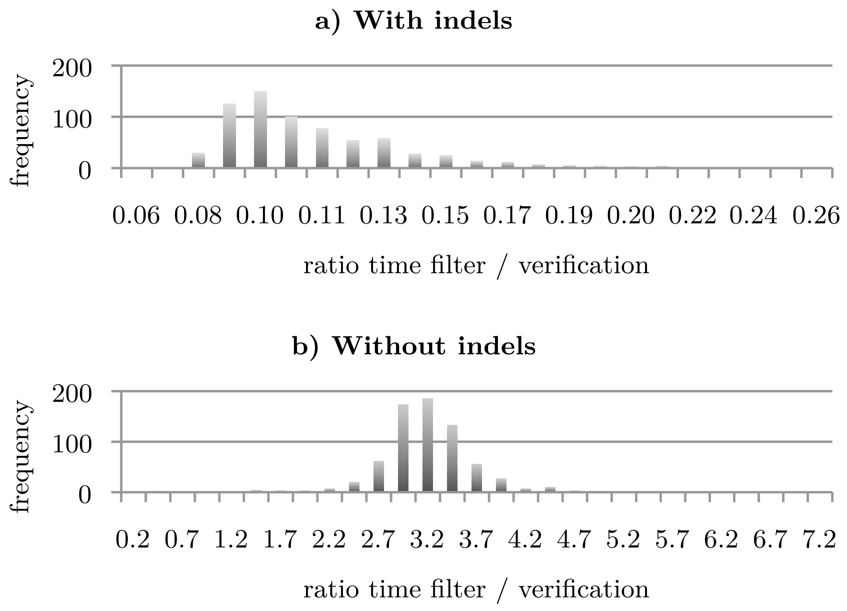


Figure 3.5: Histogram of the ratio between time for filtering 100 kilo bases reference sequence and verifying the resulting hits. Both experiments used the same data: 100.000 reads of length 100 mapped to human chromosome 22 allowing for 8 differences, a) including insertions and deletions, and using ungapped q-grams, or b) only mismatches with gapped q-grams. In the first case, the filter on average is ten times faster than the verifier. If indels are not allowed, this proportion reverses to about 3 to 1 in favor of the verifier.

As one can see from this equations, q has a big impact on the size of the index and with its growth the portion that the directory table contributes becomes dominant (see also figure 4.3, a).

The parallelization by reads described in section 3.2.2 builds multiple q-gram indices over subsets of reads instead of one over all. As a consequence the directory table also exists multiple times. However, for increasing values of q these tables are not filled completely (if $\sigma^q > n-q+1$). For $q = 16$, q-grams in ten million reads of length 100 can have maximal $10^7 \cdot 84 = 840,000,000$ of the possible $4^{16} = 4,294,967,296$ distinct hash values. If we assume that large parts of the genome are repetitive regions, this ratio grows further, as well as for longer q-grams.

3.3.1 Constructing the index

We introduce an open addressing q-gram index, which decouples the size of the directory table from the value of q. It uses a secondary hash function and an additional table called buckets. The new hash function reduces the co-domain by applying the modulo operation to the original q-gram hash value:

$$hash_{\text{sec}}(Q) = hash(Q) \mod size_{\text{buckets}} \tag{3.4}$$

It is obvious that this leads to collision when the function maps two primary to the same secondary hash value. As exit strategy, we use open addressing (OA) as described by Cormen (2001, chap. 11.4). The buckets table keeps track of which secondary hash values are already taken. During the construction of the index the algorithm calls the function `requestBucket` for every q-gram in the reads. This function first calculates $hash_{\text{sec}}(Q)$. If the entry in the buckets table at this position contains Nil (empty) or the secondary hash value itself `requestBucket` returns the position. In the former case it also writes $hash_{\text{sec}}(Q)$ in the entry. If the table is occupied by a different q-gram hash value at this position, the algorithm iterates over the table with a certain step size until it finds a free spot. Then the algorithm writes $hash_{\text{sec}}(Q)$ at this shifted position. Algorithm 5 shows the pseudo code.

Algorithm 5 RequestBucket and GetBucket

B: Buckets table, Q: q-gram

```
procedure RequestBucket($B, Q$)
1:  $l_B \leftarrow$ Length($B$) $- 1$
2:  $h_1 \leftarrow$ Hash($Q$) Mod $l_B$
3:  if $B[h_1]$ = Nil then
4:      $B[h_1] \leftarrow$ Hash($Q$)
5:      return $h_1$
6:  else
7:      if $B[h_1] = h_1$ then
8:          return $h_1$
9:      else
10:         $step \leftarrow$ prime that is not a factor of $L_B$
11:         repeat
12:             $h_1 \leftarrow (h_1 + step)$ Mod $l_B$
13:         until ($B[h_1]$ = Nil) or ($B[h_1]$ = Hash($Q$))
14:         $B[h_1] \leftarrow$ Hash($Q$)
15:         return $h_1$
16:     end if
17: end if

procedure GetBucket($B, Q$)
Same as RequestBucket($B, Q$) without lines 4 and 14.
```

Building the index based on this function works in three main steps (see algorithm 6). First, the algorithm counts the q-grams and stores the values in the directory table. Subsequently, it calculates the exclusive partial sum over all values in this table. In a last step, it goes over the read sequences again and stores the positions of the q-grams in the positions table.

Cormen (2001, chap. 11.4) uses a step size of one to search for empty entries. However, this can lead to accumulations of filled entries once multiple continuous

Algorithm 6 BuildIndex

R: Reads, B: Buckets table, D: Directory table, P: Positions table

```
procedure BuildIndex($R$)
  Fill($D, 0$)
  for $Q_\mathrm{i}$ in QGrams($R$) do
    $++D$[RequestBucket($Qi$)]
  end for

  $sum_1 \leftarrow 0$
  $sum_2 \leftarrow 0$
  for $i \leftarrow 0$ to $length_\mathrm{D} - 1$ do
    $sum_2 \leftarrow sum_2 + D[i]$
    $D[i] \leftarrow sum_1$
    $sum_1 \leftarrow sum_2$
  end for

  for $Q_\mathrm{i}$ in QGrams($R$) do
    $P$[GetBucket($Q$)] $\leftarrow i$
  end for
```

ones are taken, as they are all redirected to the same entry. This behavior is called primary clustering. Instead of one, we use a prime number that is not a factor of $size_\mathrm{buckets}$. This way the algorithm does not lead to accumulations but still visits all entries to check them.

For example, assume that $size_\mathrm{buckets} = 7$, $step = 3$, and the primary hash value is zero. If all entries but the fourth one are taken, the algorithm *getBucket* visits them in the following order: 0, 3, 6, 2, 5, 1, 4.

To estimate the size for the buckets table we first count how many q-grams fit in the reads. If n is the number of reads and l is the average length of the reads, $size_\mathrm{estimate} = n \cdot (l - q + 1)$. To reduce the expected number of collisions we set the size of the buckets and directory table to $\alpha \cdot size_\mathrm{estimate}$, where $\alpha = 1.6$, an arbitrarily chosen constant.

3.3.2 Using the index

In the traditional q-gram index all occurrences of a certain q-gram Q can be retrieved by looking up the $(directory[hash(Q)], directory[hash(Q) + 1] - 1)$-range in the positions table. For the OA q-gram index we need to modify this formula slightly to $(directory[getBucket(Q)], directory[getBucket(Q) + 1] - 1)$. This works because the buildIndex algorithm calculates the exclusive partial sum over all entries in the directory table including those which do not have a corresponding item in the buckets table. Hence, they automatically point to the first entry of the next q-gram. For an example consider figure 3.6. *Buckets*[3] is empty, hence *Directory*[3] and *Directory*[4] have the same value.

If the program looks up a q-gram that is not in the index, getBucket will search until it finds an empty entry in the buckets table and return its position (p). The entries p and $p+1$ in the directory table will then point to the same entry in the positions table, providing no occurrences for this q-gram.

Using the OA q-gram index is not always the better choice. To decide which one to use, we simple compare the two sizes (see equation 3.2 and equation 3.5).

$$size_{\text{OAindex}} = 1.6 \cdot size_{\text{estimate}} \cdot 2 + size_{\text{positions}} \tag{3.5}$$

A side effect of introducing the OA q-gram index is that we can now use larger q-grams that were not possible before. With the traditional approach the directory table for $q = 20$ for example would have had a size of about 4.4 tera byte. Using higher values for q is especially important while sequencing technologies continue to produce longer reads with decreasing error rate. Section 4.4 compares the use of different sized q-grams.

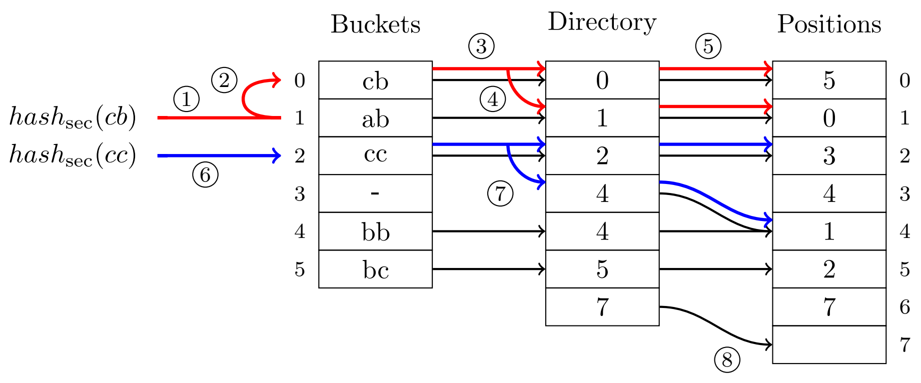


Figure 3.6: Example for an open addressing 2-gram index over the text T = "$abbcccbc$" over the alphabet $\Sigma = \{a, b, c\}$. The secondary hash function is $hash_{\text{sec}}(Q) = (hash(Q) \ \% \ size_{\text{buckets}})$, where $size_{\text{buckets}} = 6$. To access all occurrences of "cc" the algorithm (1) first calculates the secondary hash value, which is one. (2) The buckets table shows that this spot is taken by "ab". Therefore, the program uses a step size of five to iterate over the table and finds "cb" in entry zero. (3) Entry zero in the directory table (5) point to the first occurrence of "cb" in the positions table. (4) The successor of entry zero ($getBucket$("cb") + 1) points to the first entry in positions subsequent to the last occurrence of "cb". (6) Unlike for "cb", the secondary hash value of "cc" leads to entry in buckets that contains the value itself. (7) $Buckets[3]$ is empty therefore $Directory[3]$ points to the same position as $Directory[4]$. (8) In order for the algorithm to work, both directory and positions table have an empty entry at the end. Otherwise, $getBucket$("bc") + 1 would cause an error.

3.4 Implementation

This section describes the implementation details. First, in RazerS. Second, in SeqAn.

3.4.1 RazerS

Because the parallelization by reads has the least data dependencies and waiting times as the only disadvantage, we chose to implement it first. Due to its good performance and the lack of time, the other approaches were not realized.

To introduce concurrency we used OpenMP. Version 3.0 provides the task directive, which promises faster switching for nested parallelism (Lin, 2010). On the other hand, it is not supported by some compilers. However, we decided to use version 3.

We created two new parameters to control the behavior of RazerS. `-lf <num>` sets the load factor of the open addressing q-gram index. `-nc <num>` sets the number of threads that should be used to run the program. If one is specified, RazerS runs in serial mode. If the parameter is omitted, the software uses the OpenMP function `omp_get_num_procs()` to determine the optimal number of threads.

3.4.2 SeqAn

One of the objectives for this thesis is that, if changes are made to data structures from SeqAn, they still have to work outside the context of RazerS. To meet this condition we introduced the following changes to SeqAn.

The separation of the Myers' pattern from its state was realized through a new data structure called `_PatternState`. Depending on the template parameter `HasState` it is empty or contains the variables `VN`, `VP`, and `score`. The `Pattern` inherits from `_PatternState` and therefore contains the variables subject to the template parameter. If the users want to use the combined pattern, they have to specify `True` for `HasState`, or otherwise `False`. In the latter case, they also need to initialize a separate `_PatternState` and pass it to the functions associated to `Pattern` as second parameter. To retrieve the type for this corresponding `_PatternState`, we introduced the meta function `PatternState`.

However, our implementation seems to have a bug that slows it down in comparison with the original Myers' Pattern. Up to the submission of this thesis we were not able to find and remove it. Which is why the current version of RazerS still uses the sort and partition approach.

The open addressing q-gram index is implemented as a specialization of the traditional version. Using `OpenAddressing` as `TSpec` for `Index_QGram` switches it on. Since the open addressing version requires the functions `requestBucket` and `getBucket` we introduce a dummy function for the traditional index to provide a common interface.

We also created a parallel version of the functions `reverseInPlace(sequence)` and `convertInPlace(sequence, functor)`. In order to use them the user has to define the preprocessing macro `SEQAN_PARALLEL` and compile with OpenMP switched on.

Chapter 4

Results and discussion

The following chapter presents the evaluation of different approaches described above. It starts with the analysis of the methods to compensate the waiting times present in the parallelization by reads. Subsequently, we compare the performance of the parallel version with other read mappers that support multiple threads. The chapter closes with the analysis of the open addressing q-gram index and longer q-grams that are possible due to it.

We ran all following measurements on the computer described in table 4.1. We used data from the human Yoruba genome provided by Illumina. It can be accessed through the sequence read archive (http://www.ncbi.nlm.nih.gov/sra) using the experiment ID SRX016231.

Table 4.1: Specification of the computer used for the measurements presented in the results and discussion chapter.

Part	Description
Processor	2x Intel Xeon X5550
Clock speed	2.66GHz
Cores	4 per processor
Hardware threads	2 per core
Cache (level 3)	8MB shared
Cache (level 2)	256KB per core
Cache (level 1)	32KB (data) + 32KB (instructions) per core
Memory	72GB (18x 4GB)

4.1 Compensation of waiting times

The most prominent disadvantage of parallelization by reads is the difference in how long the threads need to process their subsets of reads. For the data from the Yoruba genome this disparity ranges from two to seven percent (see table 4.2). In section 3.2.2 we introduced two methods that were expected to reduce this effect. The following section examines the results of these approaches.

Table 4.2: Parallelization by reads: Waiting times of threads that finished processing the contig with their subset of reads, while other threads are still working. The measurements were done on chromosome 22 with 8 threads, different numbers of reads and verifications methods.

Verification method	Number of reads	Waiting time (total)	Run time (wall clock)	Percentage
Hamming	10^6	73.90 sec	131.63 sec	7.02%
Hamming	10^7	528.48 sec	1133.58 sec	5.83%
Edit	10^5	48.70 sec	132.03 sec	4.61%
Edit	10^6	376.49 sec	1129.90 sec	4.17%
Edit	10^7	1847.04 sec	11791.10 sec	1.96%

4.1.1 Multiple subsets of reads per thread

The first method we described creates multiple subsets of reads per thread. Because some subsets take longer, not all threads process the same number of subsets. Table 4.3 shows that the waiting times for a high number (8 x 16) of subsets in fact are lower than the originally (see table 4.2 row one). But at the same time the overall run time increases. It outweighs the improvements in the waiting time.

A closer look reveals that the verification time remains close to constant for the different parameters. The filter time on the other hand, increases with the number of blocks. The reason for this are the repeated scans of the reference sequence. Calculating the q-grams in a sequence itself is not very costly but for each subset the program looks them up in one of the q-gram indices separately. Because the q-gram hashes outside of repeat regions vary widely, the entries in the positions table of the index are on average far apart. This causes a lot of cache misses, which slows the filtering down.

4.1.2 Comparison of sorting algorithms

The second approach to compensate waiting times in the parallelization by reads redistributes the verification of hits to threads that already finished with their work. If the edit distance is chosen for the verification, the program has to separate the hits by their read IDs. The first step in this partitioning, sorts the hits.

In the fundamentals (see section 2.3) we introduced two sorting algorithms that are compared in the following. Table 4.4 shows that both algorithms perform very similar, each being better in some cases. Furthermore, smaller window sizes lead slightly better times for both algorithms. In general, the sorting takes between 1.7% to 4.1% of the time need for the verification, where a higher number of threads results in a proportional longer sorting time. This is due to the difficult balancing of work and the required communication in parallel sorting.

Because neither of the algorithms shows an advantage, we chose to use Quicksort since it is already included in the STL. However, the data also shows that avoiding the sorting step could improve the run time by up to 4% during the part of the program where the verification is redistributed to several threads.

Table 4.3: Run and waiting times for the multiple subsets per thread approach (see paragraph 3.2.2). All measurements were done on human chromosome 22 with one million of the Yoruba reads. For the verification the Hamming distance was used and the number of threads was set to eight. The runtime also includes reading in the data, building the indices, and writing out the results.

Window size	Subsets per thread	Filter time	Verification time	Waiting time (total)	Run time (wall clock)
10^4	2	759.01 sec	100.64 sec	125.28 sec	168.12 sec
10^4	4	1154.01 sec	106.27 sec	163.52 sec	223.05 sec
10^4	8	1964.08 sec	97.38 sec	115.92 sec	315.29 sec
10^4	16	3521.83 sec	98.41 sec	48.8 sec	503.86 sec
10^4	32	6281.67 sec	103.75 sec	99.44 sec	857.54 sec
10^5	2	728.39 sec	96.25 sec	114.48 sec	161.60 sec
10^5	4	1098.25 sec	95.93 sec	150.24 sec	211.08 sec
10^5	8	1884.40 sec	98.54 sec	110.8 sec	302.12 sec
10^5	16	3382.64 sec	93.29 sec	38.24 sec	475.97 sec
10^5	32	5877.19 sec	92.97 sec	64.48 sec	785.27 sec
10^6	2	721.12 sec	94.49 sec	109.04 sec	160.60 sec
10^6	4	1094.73 sec	100.39 sec	149.44 sec	211.57 sec
10^6	8	1871.15 sec	96.22 sec	112 sec	301.35 sec
10^6	16	3359.10 sec	93.01 sec	34.08 sec	472.52 sec
10^6	32	5840.69 sec	90.37 sec	53.2 sec	776.80 sec

Table 4.4: Time for sorting hits with Radix sort and Quicksort in comparison with the time needed for verifying the hits. For the measurements, 10^5 reads from the Yoruba set were mapped to chromosome 22. Both sorting algorithms show similar results, each being better in some cases. In general, the sorting takes between 1.7% to 4.1% of the time need for the verification. Smaller window sizes show a slightly better performance. The non-proportional number of hits to the window size can be explained with incomplete windows at the end of the reference sequence.

Number of threads	Window Size	Number of hits per window	Time Radix sort	Time Quicksort	Verification time
2	10^5	303082	6.222 sec	7.160 sec	364.666 sec
4	10^5	303082	4.287 sec	3.956 sec	182.368 sec
8	10^5	303082	3.677 sec	2.786 sec	91.343 sec
2	10^6	2918850	7.442 sec	7.938 sec	365.299 sec
4	10^6	2918850	4.776 sec	4.439 sec	181.908 sec
8	10^6	2918850	4.061 sec	3.103 sec	90.896 sec
2	10^7	23692596	7.383 sec	8.414 sec	366.854 sec
4	10^7	23692596	4.839 sec	5.007 sec	182.643 sec
8	10^7	23692596	3.887 sec	3.664 sec	91.198 sec

4.2 Scaling

In this section we compare the speedup of RazerS with two other read mapper - namely Bowtie and SHRiMP2. CloudBurst, which we described in the related work (see section 1.2.2), did not work, even with support from the authors.

As mentioned in the beginning of this chapter the computer used to gather the data features eight cores with two hardware threads each. Therefore, we measure the speedup for two, four, eight and 16 threads. The formula to calculate the speedup requires the runtime T_S of the fastest serial program that solves the same problem. To measure T_S we ran the read mappers with the option for one thread. Figure 4.1 and figure 4.2 show the measured speedup for hamming and edit distance respectively, as well as for different numbers of reads.

The most prominent point in the behavior of RazerS is how differently is scales depending on the number of reads that are mapped at a time. The more reads the better is the speedup. Additionally, mapping with edit rather than Hamming distance scales better as well.

A possible source of this behavior is the q-gram index. Each thread requests the occurrences in its index separately and if the number of reads is small this is prominent factor of the overall run time. A second source of overhead computation is the scheduling of the threads. However, with an increasing number of reads this extra cost becomes less important. The same is true for verification using the edit distance. It takes a lot longer than calculating the Hamming distance between read and reference sequence. Therefore the look ups in the index are a less prominent factor.

The speedup of Bowtie supports this explanation. Since it builds the index over the reference sequence the access of the index is independent from the number of threads that process the reads. Hence, the speedup is almost the same for 10^5, 10^6 and 10^7 reads. SHRiMP2 that also constructs the index over the reference sequence shows similar results. Although, it has problems with fewer reads as well.

A characteristic that is shared by all three read mappers is the comparable small improvement from 8 to 16 threads. This is due to the architecture of the computer, on which the measurements were done. While for eight threads, each can run on a separate core, this changes for 16 threads. Now one core processes two threads using hardware threads.

Hardware threads are no real parallelization. They use the time one thread wait for resources like the memory to process operations from a second one (see section 2.2.1). Therefore, the decrease in runtime we can measure from eight to 16 threads stems from using these waiting times that are avoided in the second case. But this also means that RazerS and the other read mapper spend a significant amount of time waiting for resources.

4.3 Open addressing q-gram index

The main objective for introducing the open addressing to the q-gram index was to reduce the size of the data structure for a smaller number of reads. Figure 4.3 shows

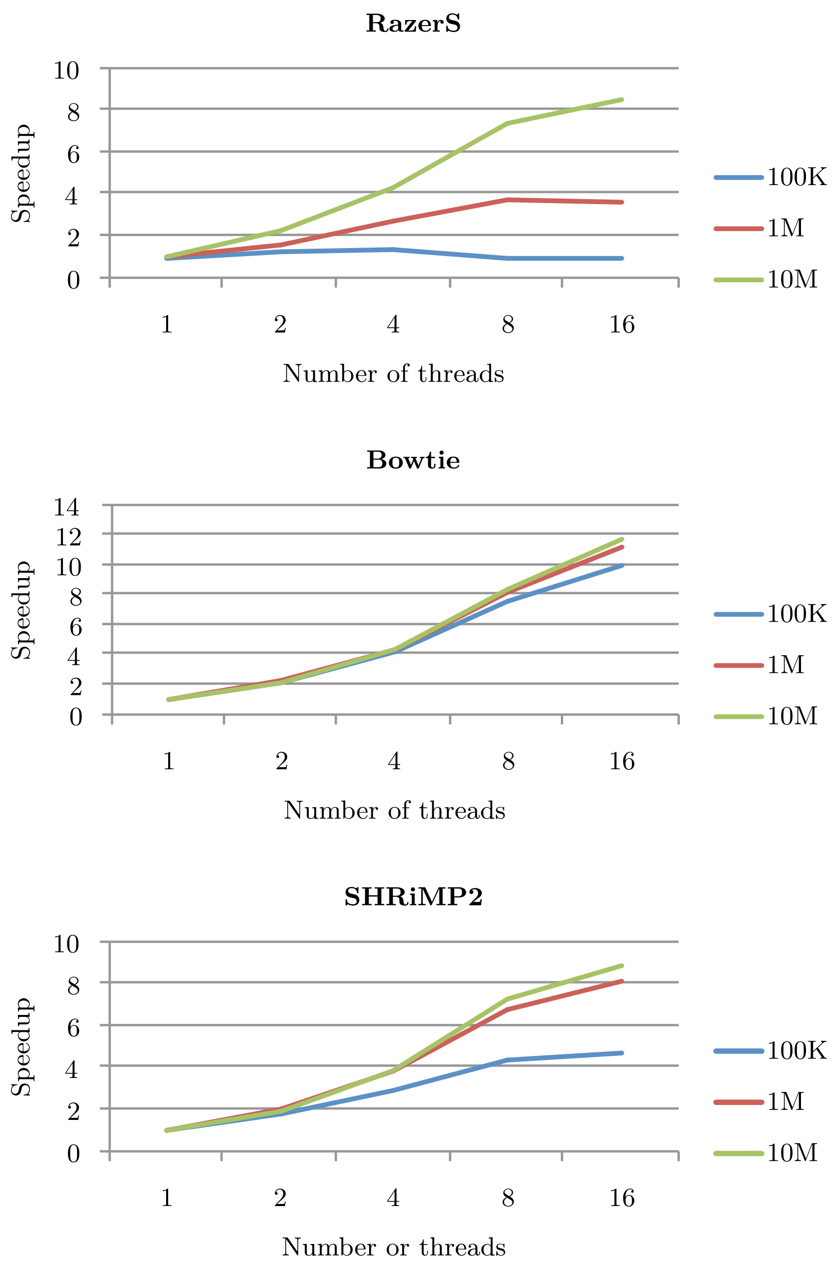
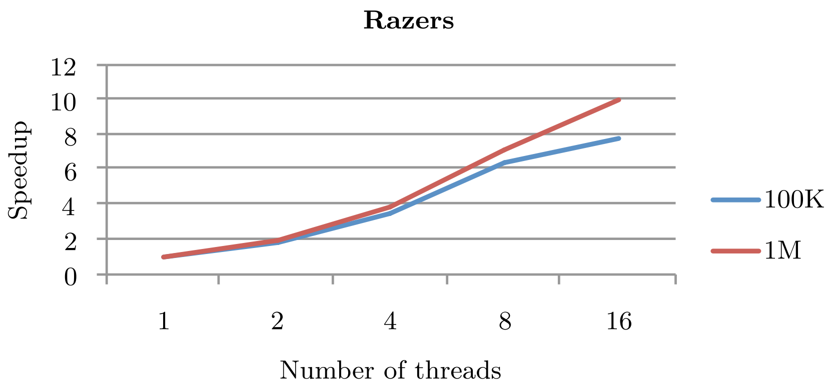


Figure 4.1: Speedup of RazerS, Bowtie, and SHRiMP2 and read sets of different size. For the measurement we mapped the Yoruba reads to human chromosome 22 using Hamming distance and allowed for 3 errors.



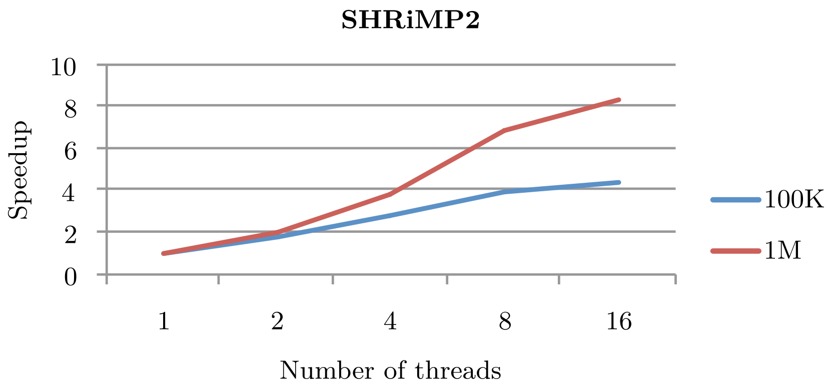
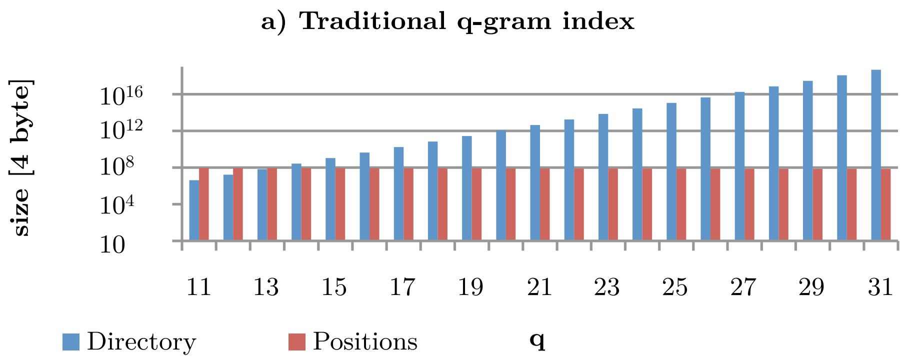


Figure 4.2: Speedup of RazerS, SHRiMP2, and read sets of different size. For the measurement we mapped the Yoruba reads to human chromosome 22 using edit distance and allowed for 8 errors. Bowtie is omitted, because it does not support edit distance.

the size for 1 million of the Yoruba reads. For $q = 14$ the total size of the new data structure is still smaller using the traditional approach. But for $q = 15$ it already exceeds the size of the open addressing index three fold.

Furthermore, the size of the open addressing index decreases for lager values of q. This is because size depends on number of positions at which q-grams start in the indexed text $(n \cdot (l - q + 1)$, where n is the number of reads and l the average length of a q-gram).



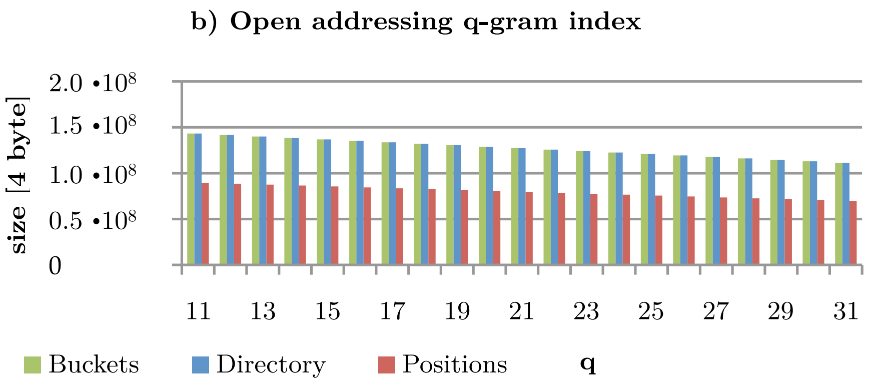


Figure 4.3: Sizes of the tables used in q-gram indices for one million (minus 5464 because of low quality) read from the Yuruba data set (SRR 034939) for different q's: a) traditional q-gram index, b) open addressing q-gram index.

The second important measure is the time consumption of the indices. This includes the time needed to build the index and to access single entries. Measurements of the former one do not show a significant difference for the two index types (about 359 seconds for 18 million Yoruba reads).

To compare the access time, we created a small program that calculates all q-gram hash values of human chromosome one (about 250 mega bases) and looks them up the indices. For five runs on average it took 8.9 and 11.8 seconds for the traditional and the open addressing index respectively. This increase of 32.6% seems high at first but if compared with the total run time of RazerS it is rather small and does not cause big deterioration of the over all performance.

A closer look at the open addressing (see table 4.5) shows that about 80% of the q-gram hash directly to a spot in the directory table. But the remaining 20% need an average of 13 steps to find an empty spot. Depending on how many steps exactly, looking up if the next possible spot is free can cause cache misses, which slows the operation down.

Table 4.5: Values for an open addressing q-gram index over one million reads and $q = 15$.

Property	Value
Size of the buckets table	136848155
Q-grams in the index	60366976
Q-grams without collisions	48035780
Q-grams with collisions	12331196
Average number steps, if collision	13.38
Number of positions	85530096

4.4 Higher Q's

In traditional q-gram indices the value of q has a direct influence on the size of the data structure, because the directory table has one entry for every possible q-gram. As consequence values of q higher than 16 or 17 are not feasible for common hardware. With the introduction of open addressing this dependency dissolves. As a result, values for q can be higher and are now restricted by the size of the variables that store the hash values. In the case of SeqAn this is 8 byte. Since the open addressing q-gram index requires one special values to mark empty spots, q can take values up to 31.

The haploid human genome has approximately three billion bases and therefore about the same number of possible different q-grams. In alphabets with four letters - like DNA - there exist 4^q different q-grams. That means under the assumption of a uniform distribution, q-grams of length 16 or more are unique in the human genome. For genetic sequences this is not entirely true but the number of occurrences per q-gram drops rapidly for lengths greater than 16.

To analyze the behavior of the RazerS with different values for q, we mapped 5 million reads of length 100 to chromosome 22, allowing 2 mismatches. We chose this number, because it allows to find all matches with q-grams up to size 31. Figure 4.4 shows four values measured.

During the filtering, RazerS counts the shared q-grams between reference and reads. The number of these counter update depends on the size of the q-grams. It drops exponentially over the whole range; from $q = 11$ to 31. This can be explained with the number of occurrences per q-gram.

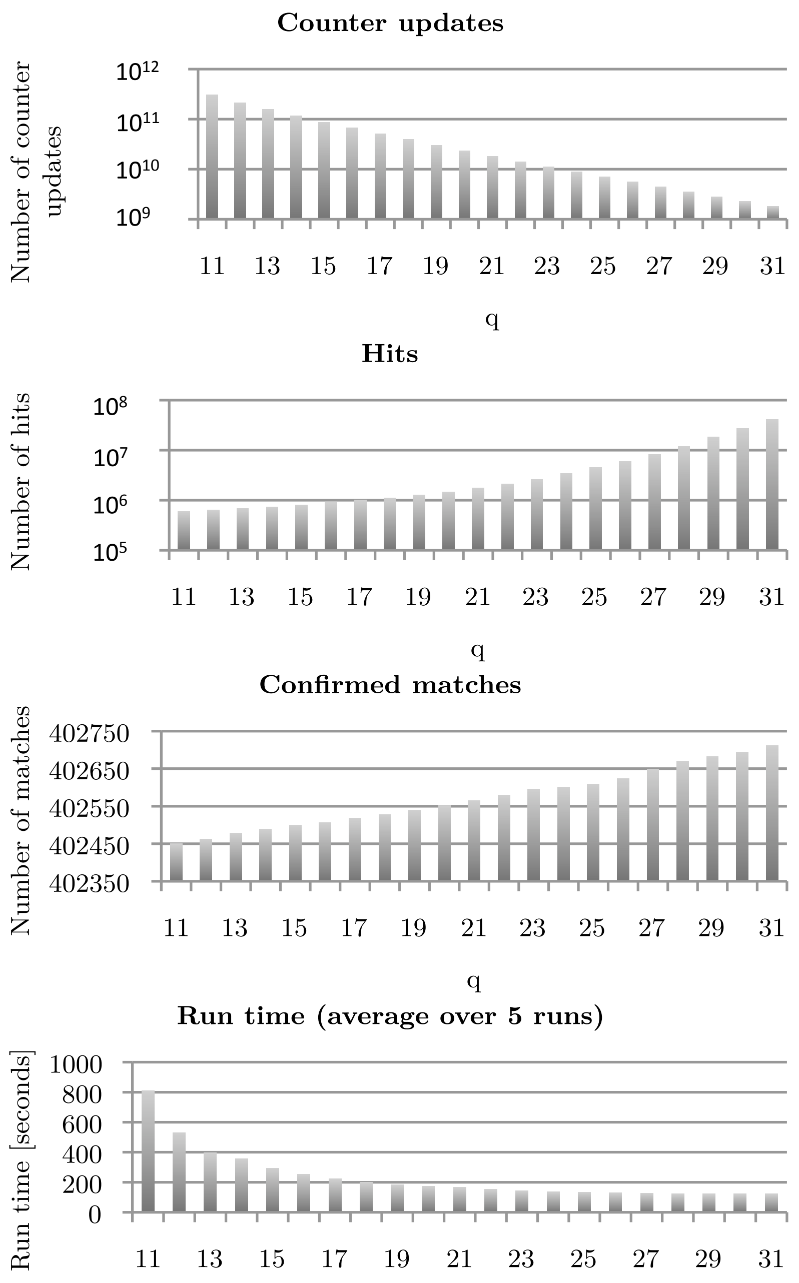


Figure 4.4: Measurement of four values for mapping with different q's. For the experiment five million reads from Yuruba were mapped to human chromosome 22 with two allows errors. The values from top to bottom are: How often the counters for shared q-grams in on band were updated; How many hits were produced; How many of these hit were confirmed and matches stored in the `alignedReadStore`; And how long each mapping took.

Shorter q-grams are less specific. Therefore, on average q-gram hash values have a higher frequency.

That means, the algorithm looks up a given hash value more often, and for each look up it increments more counter. The still falling number of updates for q-grams longer than 16 shows that DNA is not uniformly distributed and that many of the q-grams have multiple occurrences. However, they get less frequent with increasing values of q.

The hits produced by the SWIFT algorithm show the opposite behavior. With increasing values of q their number goes up as well. This is due to the minimal coverage (see lemma 2). The longer the q-grams get, the less are needed and the less sequence can be covered. This allows for more mismatches and leads to more hits.

As an example we take $q_1 = 14$ and $q_2 = 31$, a read length of 100 and two allowed mismatches. The q-gram lemma tells us that in the case of q_1 reference sequence and read share at least 59 q-grams and for q_2 only 8. Based on these values the minimum coverage is 72 (q_1) and 38 (q_2) respectively. This mean q_2 is less stringent and therefore allows for more hits.

The number of verified matches goes up slightly for larger q-grams. This as well can be explained with the minimum coverage. If a match crosses the boundary of a band due to an insertion, half of it lies in the overlap and the other part in one band only. Because higher values of q require a smaller number of matching positions, the part in the overlap can trigger a hit twice - once for each band. In contrast, smaller q-grams need additionally shared q-grams in the second half. Therefore, only the band that contains both parts will produce the hit.

The run time from 11 to 17 falls rapidly, for higher values of q it saturates, and for values over 28 there are no differences at all. An explanation for this behavior is the ratio between number of counter updates and hits that need to be verified. Small q-grams cause so many incrementations that these are the main part of the computation. For larger q-grams the contribution gets less and from $q = 19$ on the increasing number of verifications compensates this reduction and keeps the run time close to static.

Chapter 5

Conclusion

In this final chapter we are summarizing the results and give a short outlook on what is left to do and can be further improved.

5.1 RazerS

The parallelization by reads is the most commonly used with concurrent read mappers. Therefore, it was not surprising that it also showed the best trade off for RazerS.

Unlike most other read mappers, RazerS builds the index over the reads and not the reference sequence. On the one hand, this is an advantage because for the parallelization the index is split up and the parts can be calculated in parallel. On the other, it means that with the construction of the indices, the assignment of a read to a thread during the filtering is fixed and cannot be changed later on.

However, we can assume by the design of the sequencing experiments that the reads are randomized and that subsets of the same size take almost the same time to process. Table 4.2 shows that this is correct for the data used in the results section. Nevertheless, there are some waiting times and the redistribution of the verification is a good way to reduce it.

The speedup we measured for RazerS is similar to the results of Bowtie for a higher number of reads. It is close to linear while the number of threads does not exceed the number of processing cores. And even if two thread run on one core using hardware threads, we could measure a significant speedup. SHRiMP2, the second software we compared with, shows a smaller speedup than the other two read mappers.

Also the results look promising, there is still room for improvement and some future work to be done:

First, during the last phase of RazerS, the program does some final sorting and depending on the options calculates complete alignments for the output. Both steps can be parallelized. Also the writing to the file itself could be optimized, if one thread would continuously access the hard drive while other perform some additional calculations. That way the speed of the main storage could be fully utilized.

Second, because of time issues we were not able to implement the redistribution of the verification for the paired-end tag version of RazerS yet. However, because of the dependency between right and left mate, it is difficult to modify the approach of redistribution to work with the paired-end tag version. Additionally, `_PatternState` is need for this and has to be fixed.

5.2 Open addressing q-gram index

The new open addressing q-gram index is significantly slower than its traditional counterpart. However, since calculating the hash values and looking up the occurrences of the q-grams is a rather small part of the algorithm, the effect on the overall run time is minimal. Furthermore, the OA q-gram index has a real advantage for q-gram longer than 14. Giving the still growing number of reads that can be produced in one sequencing experiment, this might prove beneficial in future.

Nevertheless, the hash function of is a point where more work can be done. In the current implementation, the occurrences of adjacent q-gram are normally stored in distant parts of the memory. Therefore, it would be beneficial modify the hash function in a way that q-grams that are shifted by only one base have a similar hash value. This would lead to less cache misses since they are accessed subsequently.

The open addressing q-gram index has the advantage that it can manage higher values for q than the traditional one. We showed that runtime is shorter the higher were used. However, the use of such long q-grams requires a low error rate. But as the sequencing technology advances the quality of the reads improves as well.

Bibliography

Alkan, C., J. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. Kitzman, C. Baker, M. Malig, O. Mutlu, et al. (2009). Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature genetics*.

Altschul, S. F., T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research 25*(17), 3389.

Cormen, T. H. (2001). *Introduction to algorithms.* MIT Press.

Dean, J. and S. Ghemawat (2008). Map reduce: Simplified data processing on large clusters. *Communications of the ACM-Association for Computing Machinery-CACM 51*(1), 107–114.

Döring, A., D. Weese, T. Rausch, and K. Reinert (2008). SeqAn an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics 9*(1), 11.

Eaves, H. L. and Y. Gao (2009, April). MOM: maximum oligonucleotide mapping. *Bioinformatics 25*(7), 969–970.

Fullwood, M. J., C. Wei, E. T. Liu, and Y. Ruan (2009, April). Next-generation DNA sequencing of paired-end tags (PET) for transcriptome and genome analyses. *Genome Research 19*(4), 521–532.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press.

Hagmann, J. (2009). GenomeMapper Graph-based mapping tool for short reads from next generation sequencing data.

Hoare, C. (1962). Quicksort. *The Computer Journal 5*(1), 10.

Hoffmann, S., C. Otto, S. Kurtz, C. Sharma, P. Khaitovich, J. Vogel, P. Stadler, and J. Hackermüller (2009). Fast mapping of short sequences with mismatches, insertions and deletions using index structures.

Homer, N., B. Merriman, and S. F. Nelson (2009, November). BFAST: an alignment tool for large scale genome resequencing. *PLoS ONE 4*(11), e7767.

Illumina Inc. (2010). Every genome - sequencing FAQs. www.everygenome.com.

Jiang, H. and W. H. Wong (2008, October). SeqMap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics 24*(20), 2395–2396.

Kim, Y. J., N. Teletia, V. Ruotti, C. A. Maher, A. M. Chinnaiyan, R. Stewart, J. A. Thomson, and J. M. Patel (2009, June). ProbeMatch: rapid alignment of oligonucleotides to genome allowing both gaps and mismatches. *Bioinformatics 25*(11), 1424–1425.

Kumar, V., A. Grama, A. Gupta, and G. Karypis (1994). *Introduction to parallel computing: design and analysis of algorithms.* The Benjamin/Cummings.

Langmead, B. (2010, May). Direct correspondence.

Langmead, B., C. Trapnell, M. Pop, and S. Salzberg (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology 10*(3), R25.

Li, H. and R. Durbin (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics 26*(5), 589.

Li, H., J. Ruan, and R. Durbin (2008, November). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research 18*(11), 1851–1858.

Li, R., C. Yu, Y. Li, T. W. Lam, S. M. Yiu, K. Kristiansen, and J. Wang (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics 25*(15), 1966.

Lin, H., Z. Zhang, M. Q. Zhang, B. Ma, and M. Li (2008). ZOOM! zillions of oligos mapped. *Bioinformatics 24*(21), 2431.

Lin, Y. (2010, July). Comparing nested parallel regions and tasking in openmp 3.0. http://wikis.sun.com/pages/viewpage.action?pageId=38210769.

Maxam, A. and W. Gilbert (1977, February). A new method for sequencing DNA. *Proc. Natl. Acad. Sci. U.S.A. 74*(2), 560–4.

Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM 46*(3), 395–415.

OpenMP API 3.0 (2008). OpenMP application program interface - version 3.0.

OpenMP.org (2010, May). OpenMP.org - OpenMP compilers. http://openmp.org/wp/openmp-compilers/.

Rasmussen, K. R., J. Stoye, and E. W. Myers (2006). Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of Computational Biology 13*(2), 296–308.

Roche Applied Science (2010, May). GS FLX titanium series. http://454.com.

Rumble, S. M., P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno (2009, May). SHRiMP: accurate mapping of short color-space reads. *PLoS Comput Biol 5*(5), e1000386.

Schatz, M. C. (2009, June). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics 25*(11), 1363–1369.

Shendure, J., G. J. Porreca, N. B. Reppas, X. Lin, J. P. McCutcheon, A. M. Rosenbaum, M. D. Wang, K. Zhang, R. D. Mitra, and G. M. Church (2005, September). Accurate multiplex polony sequencing of an evolved bacterial genome. *Science 309*(5741), 1728–1732.

Smith, A. D., Z. Xuan, and M. Q. Zhang (2008). Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics 9*(1), 128.

Sodan, A., J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh (2009). Parallelism via multithreaded and multicore CPUs. *Computer*.

Strömberg, M. and W. Lee (2009). Mosaik read alignment and assembly program.

The Valgrind Developers (2010, July). Cachegrind: a cache and branch-prediction profiler.

Weese, D., A. K. Emde, T. Rausch, A. Döring, and K. Reinert (2009). RazerS—fast read mapping with sensitivity control. *Genome Research 19*(9), 1646.