UiO : Department of Informatics
University of Oslo

Improving optimal sequence alignments through a SIMD-accelerated library

Jakob Tobias Frielingsdorf

Master's Thesis Spring 2015

Abstract

The recent years have seen an increasing demand in fast sequence alignments, fuelled by a rapidly growing amount of sequence data. Meanwhile, with growing computing power optimal sequence alignment algorithms came back into the focus of these analyses.

This work presents a new library for fast database searches based on optimal sequence alignments. It performs database searches accelerated on multiple threads and single instruction multiple data (SIMD) operations. The library implements Rognes' approach for accelerating database searches, while being designed for extensibility. A modular structure allows for an easy integration of new and improved algorithms. Additionally, the application programmable interface (API) of the library is designed for easy use and flexibility, allowing an extensive configuration of the computations.

Besides the modular structure, the key features are the database searches based on SIMD instructions. These are optimised for the widely used streaming SIMD extensions (SSE) and the more recent advanced vector extensions (AVX) implementing twice as wide registers.

The focus of this thesis is the evaluation of the performance of libssa and of the optimised implementation of the database searches, with emphasis on the benefits of the computation on AVX over SSE. It presents that AVX significantly improves the performance by up to 1.83 times over SSE.

Acknowledgements

I would like to thank my supervisor Torbjørn Rognes for a lot of interesting discussions and valuable feedback which helped to improve this work.

Lukas and Olli deserve a special thanks for proofreading parts of my thesis. Both provided me with useful comments. Another big thanks goes to my friends and family for moral support.

On a personal note I would like to thank my dear Julia for her love and encouragement.

Jakob Tobias Frielingsdorf
University of Oslo
May, 2015

Contents

1 Introduction **1**

2 Background & Theory **3**

2.1 Biological background . . . 3
2.2 Sequence alignment . . . 4
2.2.1 Applications of sequence alignment . . . 5
2.2.2 Sequence translations . . . 5
2.2.3 Algorithms for sequence alignment . . . 6
2.3 Parallelisation of sequence alignment . . . 15
2.3.1 Flynn's Taxonomy . . . 15
2.3.2 Levels of parallelisation . . . 15
2.3.3 CPU . . . 16
2.3.4 Parallelising sequence alignments on CPUs . . . 17
2.4 API design in C . . . 18
2.4.1 General design rules . . . 19
2.4.2 Design rules for implementations in C . . . 20
2.4.3 Existing sequence alignment APIs . . . 21
2.5 Measuring performance . . . 23
2.5.1 Measuring speed . . . 23
2.5.2 Amdahl's Law . . . 23
2.6 Testing . . . 24

3 Design **27**

3.1 Design of the API . . . 27
3.1.1 Use cases . . . 27
3.1.2 Alignment API . . . 28
3.1.3 Database API . . . 30
3.2 Design of the library . . . 30
3.3 Parallelisation of sequence alignment . . . 32
3.3.1 Threads . . . 32
3.3.2 Non vectorised sequence alignments . . . 33
3.3.3 Vectorised alignments . . . 33

4 Implementation **37**

4.1 Configuration and internal formats . . . 37
4.1.1 Data types . . . 37
4.1.2 Sequences . . . 38

4.1.3 Scoring schemes . . . 39
4.1.4 Gap penalties . . . 40
4.1.5 Validation of the configuration . . . 40
4.2 Database integration . . . 41
4.3 Controlling database searches and alignments . . . 41
4.3.1 Thread pool . . . 42
4.3.2 Min-max-heap . . . 42
4.4 Database searches . . . 43
4.4.1 64 bit implementation . . . 44
4.4.2 8 and 16 bit implementations . . . 45
4.5 Computing alignments . . . 54
4.6 Testing . . . 55
4.7 Measuring performance . . . 56

5 Results and discussion 59
5.1 Evaluating performance . . . 59
5.1.1 Collecting results . . . 59
5.1.2 Base test run . . . 61
5.1.3 Query lengths, bit widths, and SIMD capabilities . . . 63
5.1.4 Thread counts . . . 69
5.1.5 Chunk sizes . . . 71
5.1.6 Comparison of alignment tools . . . 74
5.1.7 Tests . . . 77
5.2 Evaluating optimisations . . . 78
5.2.1 Search columns . . . 78
5.2.2 Memory management . . . 80
5.2.3 Optimisations done by the compiler . . . 82
5.3 Exploiting open source . . . 85

6 Future work 89
6.1 New and improved algorithms . . . 89
6.1.1 Needleman-Wunsch-Sellers algorithm . . . 89
6.1.2 Different gap penalties . . . 89
6.1.3 Optimising alignments . . . 89
6.2 Error handling . . . 90
6.3 Creating the temporary score profile . . . 91
6.3.1 Compute on 8 bit . . . 91
6.3.2 Different implementation for nucleotide sequences . 92
6.3.3 Filling for only match/mismatch values . . . 92
6.4 Other work . . . 92

7 Conclusion 95

Bibliography 97

A Number of search columns for different databases 99

B Reproducing performance results 101

C Installing and running libssa **103**

Chapter 1

Introduction

Sequence alignments are an important part of the analysis of genomic data. They help to discover similarities in genomic sequences and form a base for further analyses and research. Especially in the recent years the cost for sequencing genomic and protein data is as low as ever before. The effect is a rapidly growing amount of sequence data fuelling the demand for fast analyses.

Algorithms for optimally solving the problem of sequence alignments exist since the 1970s. In the past, the biggest challenge of these algorithms was their quadratic complexity in time and memory. A solution were heuristic algorithms solving the problem good but not optimally.

One of the applications based on sequence alignments are database searches. Here, a database is searched for sequences, that are similar to a query sequence. Since the beginning of the 1990s heuristics were the main kind of algorithms used for such applications.

With growing computing power and more advanced hardware, optimal sequence alignment algorithms came back into focus. One technique that is used to speed up these algorithms are single instruction multiple data (SIMD) operations. CPUs implement this feature since the middle of the 1990s. It allows for computing on multiple data elements in parallel and was used to develop different approaches to accelerate optimal sequence alignments.

In software development it is common practice to implement tasks in libraries, which are then used in multiple programs. This allows for re-using an implementation of a task in different programs, thus speeding up the development of these. This thesis describes a new library for database searches using optimal sequence alignment algorithms called libssa (library for SIMD accelerated optimal Sequence Alignments). It implements the searches based on global and local alignments using the Needleman-Wunsch and Smith-Waterman algorithms. The focus of the library lies on fast implementations of the database search and algorithms as well as on providing a modular structure, which allows for an easy extension of the library.

Two of the SIMD instruction sets, which implement the SIMD operations in modern CPUs, are the streaming SIMD extensions (SSE) and the advanced vector extensions (AVX). The main difference of both is the amount of data elements, which can be computed in parallel. AVX allows for twice as many data elements than SSE. The research question of this thesis is whether an implementation of the optimal sequence alignment algorithms based on AVX instructions performs better than an implementation based on SSE instructions. In addition to the quantitative analysis of the library's performance, it is going to be compared to existing implementations of database searches.

Chapter 2

Background & Theory

2.1 Biological background

The entire genetic information of an organism is stored in the genome. Encoded in DNA (Deoxyribonucleic acid), it describes the features and properties of the organism. DNA is a molecule which consists of two complementing nucleotide sequences that form a double helix. This structure is formed by bonds between the nucleotides of the complementing sequences. The four different nucleotides or nucleobases, which DNA is build of, are: adenine (A), guanine (G), cytosine (C), and thymine (T). Adenine and guanine, and cytosine and thymine are complementary to each other.

The structure of cells and nearly all cell functions in the body of an organism are built of or executed by proteins. The blueprints for building proteins are encoded in genes, the coding entities of the genome. These are transcribed into RNA (Ribonucleic acid), an intermediary molecule similar to DNA, which guides the synthesis of protein molecules. RNA is build of the nucleotides adenine (A), guanine (G), cytosine (C) and uracil (U). Uracil substitutes the thymine of DNA. The RNA then are translated into a chain of amino acids, which forms a protein. The translation from RNA to amino acids is described in the genetic code (see figure 2.1). Each of the 20 amino acids that occur in an organism is described by a triplet of RNA nucleotides, a so called codon. This code is universal for all organisms. [Sung, 2010, chapter 1]

The actual process of the protein biosynthesis is more complex than described here. The focus here was to introduce terms used in the next chapters, which describe different approaches and algorithms for sequence alignments.

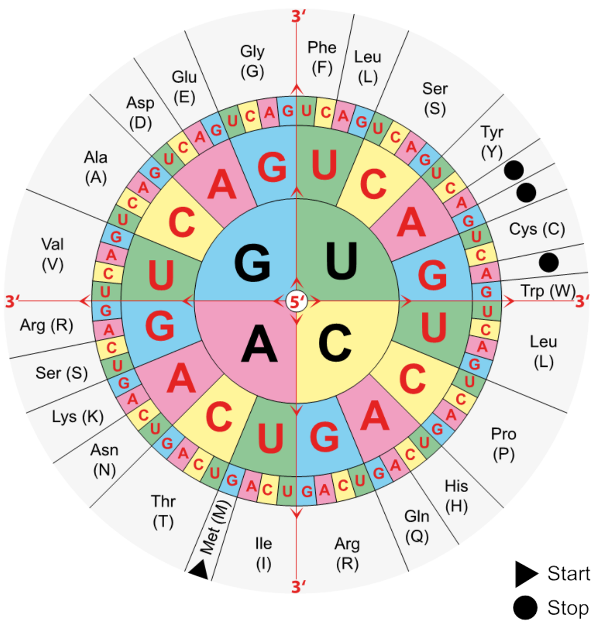


Figure 2.1: The figure shows the genetic code for the translation from RNA bases to amino acids. The black triangle marks the start codon, while the black circles mark the stop codons. Both indicate the beginning and end of genes in the DNA. Source: Wikimedia - https://commons.wikimedia.org/wiki/File:Aminoacids_table.svg

2.2 Sequence alignment

Sequence alignments are used to explore the evolutionary or functional relations of genomic sequences. They are tools to measure similarity between sequences, which can give information about common ancestors or similar functions. The important issues here are the biological relevance, and the speed of the alignments.

The biological relevance is an issue, since the similarity is only an indicator of the similar functions of two sequences. The functional similarity has to be confirmed later by experiments.

The speed of an alignment on the other hand is important, since genomic sequences can be either very long or the amount of sequences to compare is very high. Sung illustrates this in his book in the following way: "using a 3.0 GHz PC with 4GB RAM, it takes more than 15 hours to align a sequence of 1000 nucleotides with the whole human genome", using the Smith-Waterman optimal alignment algorithm [Sung, 2010, p. 111]. The human genome, which has about 3 billion base pairs, is more of an average sized genome. The smallest known genome of a free living organism has about 600 000 base pairs (*Mycoplasma genitalium*), in contrast to the largest known genome with about 670 billion base pairs (*Amoeba dubia*) [Sung, 2010, p. 10].

2.2.1 Applications of sequence alignment

Sequence alignments are used in a variety of genomic research and analyses. This section gives an overview of common applications based on sequence alignments to illustrate what the algorithms described in section 2.2.3 are used for.

Database searches One of the most widely used applications is to search for similar sequences in a sequence database. These databases contain nucleotide or protein sequences of multiple different species. The result of such a search is a list of sequences, which are similar to the query sequence. This can help to identify the functions of unknown sequences assuming that similar sequences have a similar function.

Current databases contain millions of different sequences, with its number growing rapidly. In 2008 for example, the International Nucleotide Sequence Databases (INSD)[1] contained 110 million sequences with 200 million base pairs for more than 204 000 named organisms. Current numbers for GenBank, one of the databases collaborating in INSD, are found on their website[2]. [Sung, 2010, p. 109]

Clustering Another application is the clustering of genomic sequences into operational taxonomy units (OTU). Here the sequences are clustered based on their similarity.

Sequence assembly Current sequencing technologies cannot read genomic sequences of more than a couple of hundred base pairs. A solution to this is to multiply the sequence first by PCR, cut these into small pieces, sequence them, and then assemble these pieces to get the continuous sequence. The assembly of the small pieces, or short reads, often involves alignments. The alignments indicate overlapping reads, which are assembled to longer reads, to get the whole sequence.

Short read mapping The task of short read mapping is similar to the task of sequence assembly: the sequencing and reading of a genomic or protein sequence. The difference here is that a reference sequence has already been read. The short reads can then be aligned to the reference sequence, to find their position on the longer reference sequence.

2.2.2 Sequence translations

Proteins are build from the information encoded in the genes, of an organisms DNA. On a simple level, the nucleotides of a gene are translated to the amino acids of a protein. The translation is done using the genetic code. These translations can be done manually to align a nucleotide to a protein

[1] http://www.insdc.org, last visited 10.04.2014
[2] https://www.ncbi.nlm.nih.gov/genbank/statistics, last visited 10.04.2014

sequence. The nucleotide sequence is then translated and both sequences are aligned as protein sequences. [Sung, 2010, chapter 1]

The translations are done using the genetic code, shown in figure 2.1, where a triplet of nucleotides is translated to an amino acid. The translations are done in up to 3 reading frames and up to 2 strands.

The 2 strands are the complementary strands DNA is made of. This is resembled with the translation of the nucleotide sequence done twice. Once as it is provided and once as the reverse complement, mimicking the second strand of the DNA. RNA occurs usually as only one strand. Therefore only the provided strand is translated.

The 3 reading frames are used since it is often not known at which position, of the nucleotide sequence, the protein sequence starts. Figure 2.2 shows how the frames are used, to translate the RNA sequence into the 3 possible protein sequences.

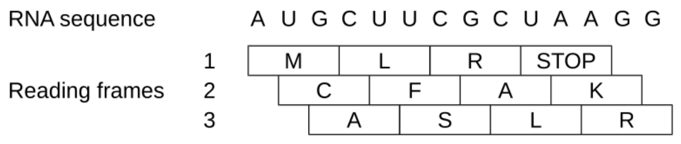


Figure 2.2: Shows the translation of a RNA sequence to all 3 reading frames, using the genetic code. The amino acids are shown in the one letter representation, with STOP being a stop codon.

The translation of a nucleotide sequence results in up to 6 possible amino acid sequences. These are then aligned to amino acid sequences.

2.2.3 Algorithms for sequence alignment

Based on the applications described in section 2.2.1 there are different kinds of sequence alignments. Each kind is based on an algorithm that returns the best result for it. The focus here is on optimal algorithms, which are later to be implemented by the API. Therefore algorithms based on heuristics or filters are not mentioned.

The base for the different kinds of sequence alignments is formed by the string edit problem. It computes the minimum number of operations to transform one string into another. The operations are:

- Insertion of a symbol
- Deletion of a symbol
- Substitution of a symbol

In sequence alignments, an insertion or deletion is a gap in the alignment of two sequences. A gap in the first sequence is called an insert while a gap in the second sequence is called a delete. If two aligned symbols are the same it is called a match, otherwise it is called a mismatch, which is equal to a substitution. [Sung, 2010, p. 30, 31]

Using these operations, the edit distance for the strings or sequences *abc* to *abbd* would be 2: one insertion of the symbol *b* and one substitution of the symbol *c* with *d*. One possible alignment of the two sequences looks as follows:

```
Sequence 1: ab-c
            ||
Sequence 2: abbd
```

The two vertical bar symbols denote a match, the inserted symbol *b* results in a gap in sequence 1, and the last position is a mismatch. Another possible alignment would be a mismatch at position 3 and an insertion of the symbol *d*, at position 4. Often there are multiple ways to transform one string into another, with an equally short distance. This is behaviour is common to sequences alignments as well and in both all alignments or conversions, with the same distance, are equally good.

Another representation of sequence alignments are CIGAR strings (Compact Idiosyncratic Gapped Alignment Report). These describe the positions and lengths of deletions, insertions, and substitutions. The CIGAR string for the alignment above is `2M1I1M`: 2 matches, 1 insertion, and 1 mismatch. A compressed form are CIGAR strings omitting the count if it is only 1: `2MIM`. [Li et al., 2009]

Global sequence alignments

Global sequence alignments are used to align entire sequences. They basically implement the string edit problem, calculating all possible transformations from sequence A to sequence B, while keeping the transformation with the lowest distance as alignment. The difference lies in the calculation of the score. Instead of adding up the operations to calculate the distance, the score is increased in case of a match, while mismatches and gaps are penalised. The result is a score indicating similarity, where a higher score implies a higher similarity. [Sung, 2010, p. 30ff]

Needleman-Wunsch: The most commonly used optimal alignment algorithm for global alignments is the Needleman-Wunsch algorithm [Needleman and Wunsch, 1970]. It is a dynamic programming algorithm that first calculates a matrix with the optimal alignment scores. In a second step it calculates the optimal alignment using the matrix. [Sung, 2010, p. 32]

This part describes the Needleman-Wunsch algorithm with linear gap costs only, while later other gap costs are described. Linear gap costs define the penalty for a gap to be linear to the length of the gap. Hence the penalty for a gap of length n with costs R is $g = n * R$.

The following describes the computation of the alignment scores in the matrix H. The first row and column, for $i = 0$ and $j = 0$, are initialised with the scores for aligning a sequences with an empty sequence, like shown in equation 2.1. The variable i is the index in sequence A, while j is the index in sequence B.

$$H_{0,0} = 0, \quad H_{0,j} = H_{0,j-1} + R, \quad H_{i,0} = H_{i-1,0} + R \tag{2.1}$$

Each cell, of the alignment matrix, is calculated based on previous cells. Hence the equations for each cell are defined recursively, based on the previous cells.

The cells starting at index $1, 1$ are calculated using equation 2.2. The value of the cell, at index i, j, is calculated as the maximum of a match or mismatch, a deletion, and an insertion. An insert adds the gap costs R to the value in the left cell. A delete adds the gap costs R to the value in the upper cell. While a match or mismatch adds the value for a match or mismatch to the upper left cell. The value for a match or mismatch is calculated using the function $V(i, j)$, which returns the value for a match, if the symbols in both sequences at the indices i and j match. Otherwise the function returns the mismatch costs. This is done until the values for all cells are computed.

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} & +V(i,j) & \text{match/mismatch} \\ H_{i-1,j} & +R & \text{insert} \\ H_{i,j-1} & +R & \text{delete} \end{cases} \tag{2.2}$$

Table 2.1 shows an example of the alignment matrix calculated for the sequences `TACGGGTAT` and `GGACGTACG`. The alignment values are computed using gap costs of $R = -1$, a match value of 1, and a mismatch value of -1. The score for the optimal global alignment is obtained from the cell at the bottom right corner of the matrix, which is -1.

The optimal alignment is gained by backtracking from the bottom right corner to the top left corner. To reduce computing time for the alignment, it is common practice to save the directions in a second matrix, like shown in table 2.2. The value in each cell is a bitmap. The right most bit, at index 1, encodes an insertion, the second bit encodes a deletion, and the third bit encodes a match or mismatch. A set bit states that a cell was reached through the encoded action.

The global alignment obtained from table 2.2 is the following:

```
Sequence 1: -TACGGGTA-T
              ||  |||
Sequence 2: GGAC--GTACG
```

		G	G	A	C	G	T	A	C	G
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
T	-1	**-1**	**-2**	-3	-4	-5	-4	-5	-6	-7
A	-2	-2	-2	**-1**	-2	-3	-4	-3	-4	-5
C	-3	-3	-3	-2	**0**	-1	-2	-3	-2	-3
G	-4	-2	-2	-3	-1	**1**	0	-1	-2	-1
G	-5	-3	-1	-2	-2	**0**	0	-1	-2	-1
G	-6	-4	-2	-2	-3	**-1**	-1	-1	-2	-1
T	-7	-5	-3	-3	-3	-2	**0**	-1	-2	-2
A	-8	-6	-4	-2	-3	-3	-1	**1**	**0**	-1
T	-9	-7	-5	-3	-3	-4	-2	0	0	**-1**

Table 2.1: Matrix showing the alignment score for the global alignment of the sequences: `TACGGGTAT` and `GGACGTACG`. The cells in bold face where used to gain the optimal alignment.

		G	G	A	C	G	T	A	C	G
	000	001	001	001	001	001	001	001	001	001
T	010	100	101	101	101	101	100	001	001	001
A	010	110	100	100	001	001	001	100	001	001
C	010	110	110	010	100	001	001	001	100	001
G	010	100	100	011	010	100	001	001	001	100
G	010	110	100	001	010	110	100	101	101	100
G	010	110	110	100	111	110	110	100	101	100
T	010	010	010	110	100	010	100	001	101	010
A	010	010	010	100	001	010	010	100	001	001
T	010	010	010	010	100	111	110	010	100	101

Table 2.2: Matrix showing the directions for the global alignment of the sequences: `TACGGGTAT` and `GGACGTACG`. The directions are encoded as set bits. Bit 1 set is a gap in the first sequence, bit 2 set is a gap in the second sequence, and bit 3 set is a match or mismatch. To keep it simple, only the lower three bits are shown.

Needleman-Wunsch as described by Sellers: Optimal global alignments are often described as a problem of maximising the alignment score. A high score implies a high similarity. Another way to describe similarity, is to define it by the edit-distance like in the original string-edit problem. In 1974 Sellers [Sellers, 1974] showed that similarity defined by the edit-distance is equivalent to the similarity defined by a maximised score.

Using the edit-distance is done by exchanging the scoring system in the Needleman-Wunsch. The original Needleman-Wunsch decreases the score for mismatches and gaps, and increases it for matches. The Needleman-Wunsch using Sellers scoring system increases the score, or cost for mismatches and gaps and adds 0 for matches. This results in a score of 0 for two identical sequences, in contrast to a high score implying a high diversity.

Local sequence alignments

Local sequence alignments are used to find similar nucleotide or protein sequences in a database. It aligns a sequence against another one ignoring gaps in the beginning and end, and focusing on regions with a high similarity. Like the global alignment problem the local alignment problem is also based on the string edit problem. The differences are the scoring functions and the finding of the optimal alignment. [Sung, 2010, p. 39ff]

Smith-Waterman: The Smith-Waterman algorithm [Smith and Waterman, 1981] is a variation of the Needleman-Wunsch algorithm. It computes a local alignment of two sequences. Similar to the Needleman-Wunsch algorithm the Smith-Waterman first calculates the optimal alignment scores in a matrix. In a second step it computes the optimal local alignment using the matrix. [Sung, 2010, p. 39ff]

The differences between both algorithms are the initialisation of the first row and column, the computation of the scores, and the position of the optimal alignment score. The description here is reduced to linear gap costs, like the description of the Needleman-Wunsch algorithm. Other gap costs are described further down.

In the Smith-Waterman the alignment score, for aligning an empty sequence to a non empty sequence, is 0. Hence the first row and column are initialised with 0, like shown in equation 2.3.

$$H_{0,0} = 0, \quad H_{0,j} = 0, \quad H_{i,0} = 0 \tag{2.3}$$

The rest of the cells is calculated in the same order like in the Needleman-Wunsch algorithm, from the top left corner to the bottom right corner, starting at index $1, 1$. Equation 2.4 shows the computation of the cells. The value of the cell, at index i, j, is calculated as the maximum of 0, a match or mismatch, a deletion, and an insertion. An insert adds the gap costs R to the value in the left cell. A delete adds the gap costs R to the value in the upper cell. While a match or mismatch adds the value for a match or mismatch to the upper left cell. The value for a match or mismatch is calculated using the function $V(i, j)$, which returns the value for a match, if the symbols in both sequences at the indices i and j match. Otherwise the function returns the mismatch costs. This is done until the values for all cells are computed.

The difference to the Needleman-Wunsch algorithm is the 0, as a fourth parameter of the maximum function. This parameter reduces the impact of too many mismatches and gaps on the alignment score. This way areas with a higher amount of matches are highlighted.

$$H_{i,j} = max \begin{cases} 0 & & \\ H_{i-1,j-1} & +V(i,j) & \text{match/mismatch} \\ H_{i-1,j} & +R & \text{delete} \\ H_{i,j-1} & +R & \text{insert} \end{cases} \tag{2.4}$$

Table 2.3 shows an example of the alignment matrix computed for the sequences `TACGGGTAT`) and `GGACGTACG`. The alignment values are computed using gap costs of $R = -1$, a match value of 1, and a mismatch value of -1. The score for the optimal local alignment is the highest value found in the matrix: $H(4,9) = 4$.

	G	G	A	C	G	T	A	C	G
0	0	0	0	0	0	0	0	0	0
T 0	0	0	0	0	0	**1**	0	0	0
A 0	0	0	1	0	0	0	**2**	1	0
C 0	0	0	0	2	1	0	1	**3**	2
G 0	1	1	0	1	3	2	1	2	**4**
G 0	1	2	1	0	2	2	1	1	3
G 0	1	2	1	0	1	1	1	0	2
T 0	0	1	1	0	0	2	1	0	1
A 0	0	0	2	1	0	1	3	2	1
T 0	0	0	1	1	0	1	2	2	1

Table 2.3: Matrix showing the alignment score for the local alignment of the sequences: `TACGGGTAT` and `GGACGTACG`. The cells in bold face where used to gain the optimal alignment.

The optimal alignment is retrieved by backtracking from the cell with the highest value to the next cell with value 0. The rest of both sequences is discarded. Similar to the global alignment, the computing time is reduced by saving the directions in a second matrix, as shown in table 2.4. The value in each cell is a bitmap. The right most bit, at index 1, encodes an insertion, the second bit encodes a deletion, and the third bit encodes a match or mismatch. A set bit states that a cell was reached through the encoded action.

The local alignment, obtained from table 2.4 is the following:

```
Sequence 1: TACG
            ||||
Sequence 2: TACG
```

Semi-Global sequence alignments

Semi-global alignments are similar to global alignments. They are used to align two sequences in whole, but use a different scoring system for the alignments. Depending on the application, they ignore gaps in the beginning or end of the sequences. Table 2.5 shows the different kinds of

		G	G	A	C	G	T	A	C	G
	000	000	000	000	000	000	000	000	000	000
T	000	000	000	000	000	000	100	001	000	000
A	000	000	000	100	001	000	010	100	001	001
C	000	000	000	010	100	001	001	010	100	001
G	000	100	100	001	010	100	001	001	010	100
G	000	100	100	001	011	110	100	101	010	110
G	000	100	100	101	101	110	110	100	111	110
T	000	010	010	100	101	010	100	001	101	010
A	000	000	010	100	001	001	010	100	001	001
T	000	000	000	010	100	101	100	010	100	101

Table 2.4: Matrix showing the directions for the local alignment of the sequences: `TACGGGTAT` and `GGACGTACG`. The directions are encoded as set bits. Bit 1 set is a gap in the first sequence, bit 2 set is a gap in the second sequence, and bit 3 set is a match or mismatch. To keep it simple, only the lower three bits are shown.

semi-global alignments and their difference to global alignments. [Sung, 2010, p. 41, 42]

Spaces	**Action**
In the beginning of *A*	Initialise first row with 0
In the ending of *A*	Look for the maximum in the last row
In the beginning of *B*	Initialise first column with 0
In the ending of *B*	Look for the maximum in the last column

Table 2.5: Table showing actions for ignoring spaces in the beginning or end of two sequences *A* and *B*, in semi-global alignments. Source: [Sung, 2010, p. 42]

Gap penalties

Sequence alignments face the challenge of representing best the biological data, in different applications. To adapt them to different applications, sequence alignment algorithms are configured with different gap penalties.

Three commonly used gap penalties are constant, linear, and affine gap penalties. Constant gap penalties are independent from the length of the gap. Linear gap penalties grow linearly to the length of the gap and affine gap penalties apply different costs for opening and extending a gap.

The costs g for a gap with constant penalties R are $g = R$. Independent from the length. With linear gap penalties, for a gap of length n, the costs are $g = n * R$, while they are $g = Q + n * R$ for affine gap penalties. With Q being the penalty for opening a gap, and R being the penalty for extending a gap.

Sequence alignments, with constant and linear gap penalties, can be computed in time $O(n * m)$, for two sequences of length m and n. With affine gap penalties the time increases to $O(n * m * (n + m))$, since for each cell the algorithm has to check if a gap is extended or a new one is opened. In 1982 Gotoh [Gotoh, 1982] described a method to compute optimal sequence alignments, with affine gap penalties, in time $O(n * m)$. His version uses two additional matrices (E and F) to keep track of opened gaps. E keeps track of gaps in the query sequence and F of gaps in the database sequence.

Equation 2.5, 2.6, and 2.7 show the computation of the alignment scores for the Smith-Waterman algorithm with affine gap penalties using Gotoh's method. In all three matrices the first row and column, at the indices $i = 0$ and $j = 0$, are set to 0. The values in E are calculated as the maximum of the previous value in H plus the costs Q for opening a gap and the the previous value in E plus the costs R for extending a gap. The values in F are computed the same way, except for gaps in the other sequence. The values in H are computed like in the Smith-Waterman with linear gap costs, except the values in E and F are used as the gap costs.

The difference to linear gap costs are the two additional matrices and the additional penalty for opening a gap. The rest of the computations are the same.

$$H_{i,j} = \begin{cases} max \begin{cases} H_{i-1,j-1} + V(i,j) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} & \left| \begin{array}{c} i > 0 \\ \cap \\ j > 0 \end{array} \right. \\ 0 & \left| \begin{array}{c} i = 0 \\ \cap \\ j = 0 \end{array} \right. \end{cases} \tag{2.5}$$

$$E_{i,j} = \begin{cases} max \begin{cases} H_{i,j-1} + Q \\ E_{i,j-1} + R \end{cases} & \left| i > 0 \right. \\ 0 & \left| i = 0 \right. \end{cases} \tag{2.6}$$

$$F_{i,j} = \begin{cases} max \begin{cases} H_{i-1,j} + Q \\ F_{i-1,j} + R \end{cases} & \left| j > 0 \right. \\ 0 & \left| j = 0 \right. \end{cases} \tag{2.7}$$

The Needleman-Wunsch algorithm, with affine gap costs, is computed similarly to the Smith-Waterman algorithm, with affine gap costs. The differences are the initialisation of the first row and column and the 0 as a fourth parameter. In the Needleman-Wunsch the values g, in the first row and column, are initialised to $g = Q + i * R$ and $g = Q + j * R$, with i and j being the indices in the first row and column. The fourth parameter 0 is omitted, like in the Needleman-Wunsch algorithm with linear gap costs, as described earlier.

Similarity matrices:

Another way to configure alignment algorithms is to define the scores for matches and mismatches in similarity matrices. They define scores, representing the physical and functional similarity between the nucleotides and amino acids, of genomic and protein sequences.

The two mainly used types of similarity matrices for are point accepted mutation matrices (PAM) and block substitution matrices (BLOSUM). Both define a similarity for amino acids. PAM matrices are based on point accepted mutations, which either do not change the function of a protein or that are not fatal. These are empirically generated from phylogenetic trees of highly similar sequences [Henikoff and Henikoff, 1992]. BLOSUM matrices are constructed directly from observed alignments by comparing a number of divergent sequences. [Sung, 2010, p. 51, 51]

Time and memory requirements

To compute the optimal alignment of two sequences, algorithms, like the Needleman-Wunsch or Smith-Waterman, compare every residue of one sequence to every residue of the other sequence. Having two sequences of length m and n this takes at least $O(n * m)$ time, and $O(n * m)$ memory if the alignment should be returned. Since the global and local alignment problems are difficult to improve, people have tried to identify cases where they can be solved efficiently [Sung, 2010, p. 34ff].

Time: Computing an optimal alignment with linear or constant gap penalties takes $O(n * m)$ time. If the algorithm uses affine gap penalties, the computing time is at least $O(n * m * (n + m))$; for every cell in the matrix, the algorithm has to check if the previous cells horizontally and vertically already opened a gap, or if a new gap is opened. Hence the extension $(n + m)$ to the runtime. In 1982 Gotoh [Gotoh, 1982] introduced a method to reduce the runtime to $O(n * m)$. His extension are two matrices F and E, which hold the information about horizontally or vertically opened gaps.

Memory: To compute only the alignment score of two sequences the minimal memory requirements are $O(min(m, n))$. The matrix is calculated row by row or column by column. Hence only one row or column is kept in memory. If the alignment should be computed as well, the minimal space requirement is $O(n * m)$. In 1975 Hirschberg described a linear space approach to finding the longest common subsequence [Hirschberg, 1975]. This was used by Myers and Miller to develop a linear space version of Gotoh's algorithm [Myers and Miller, 1988]. Their algorithm uses a divide and conquer approach to reduce the memory requirements. The runtime is still $O(n * m)$ in theory, while in practice the runtime is about two times higher than the non space efficient optimal alignment algorithms.

Banded alignments: Another way to reduce the runtime of alignments are banded alignments. Here the maximum number of insertions or deletions is restricted. This results in an alignment algorithm which fills out only the middle band of the alignment matrix. The runtime for this variant of the Smith-Waterman or Needleman-Wunsch is $O((n + m) * d)$. For two sequences of length m and n, and a maximum number of insertions or deletions of d. [Sung, 2010, p. 34, 35]

2.3 Parallelisation of sequence alignment

One of the challenges of optimal sequence alignment algorithms is the number of calculations, which have to be processed. Algorithms like the Needleman-Wunsch and Smith-Waterman have a quadratic time complexity, making them computational demanding for large scale data processing. Parallelisation is a method to reduce the computing time by running multiple calculations in parallel.

2.3.1 Flynn's Taxonomy

In 1966 Michael J. Flynn proposed a classification of computer architectures, based on the types of information handled by a processor: instructions and data [Flynn, 1972]. Based on these two independent dimensions, Flynn's taxonomy defines four different computer architectures:

- Single Instruction, Single Data stream (SISD)
- Single Instruction, Multiple Data streams (SIMD)
- Multiple Instructions, Single Data stream (MISD)
- Multiple Instructions, Multiple Data streams (MIMD)

SISD architectures are the traditional computer architectures. They operate sequentially, using one control unit, one processing unit, and one memory unit. SIMD architectures have more than processing unit. These are used to exploit data parallelism by executing one instruction on multiple data streams at the same time. MISD architectures, on the other hand, implement multiple processing units to execute different instructions on one data stream. MIMD architectures can process different instructions on different data streams, independent from each other. These architectures implement multiple processors where each has its own control unit.

2.3.2 Levels of parallelisation

Based on Flynn's taxonomy one can define two kinds of application parallelism: Data-Level Parallelism (DLP) and Task-Level Parallelism (TLP). In DLP many data elements are processes in parallel while in TLP many tasks are processed in parallel.

The kind of application parallelism determines which kind of computer architecture is best used, for the optimal performance. An application exploiting DLP benefits most from a SIMD or MIMD architecture, which both operate on multiple data streams. On the other hand a TLP exploiting application benefits most from a MISD or MIMD architecture. Although a MISD architecture would only help, if the tasks operate all on the same data. [Hennessy and Patterson, 2012]

2.3.3 CPU

Modern CPUs implement multiple techniques to enable parallelism and to speed up the execution of programs. They implement a MIMD architecture, integrating multiple independent cores, with each being implemented as a SIMD architecture.

Each core of a CPU can run one thread or process at a time, with each of these executing its own independent task. On Intel(R) CPUs, a technique called hyper-threading[3] enables the execution of two threads simultaneously on the same core. The second thread is then executed on the execution units which are not used by the first thread. This technique can boost the overall performance, if more threads are executing than cores are available.

Intel Turbo Boost[4] is another technique to speed up the execution. It can increase the clock rate of a core, above the base operating frequency. The increased clock rate is limited by the current working load and thermal limits of the CPU, and with multiple cores working, the effect decreases.

A third technique on CPUs are SIMD operations. These are used for exploiting data level parallelism, like illustrated in figure 2.3. Here the data is loaded in vector $R1$ and $R2$, both vectors are multiplied, and afterwards stored in the vector $R3$. This processes the 4 data pairs in the vectors $R1$ and $R2$ in parallel.

SIMD operations exist for multiple purposes, like for example matrix calculations for graphics processing. Implemented are these in instruction sets, with each instruction set bundling the operations for a different purpose. Recent CPUs implement among others the MMX, SSE, and AVX instructions sets, which operate on different CPU registers of different width. MMX on the 64 bit wide MMX registers, SSE on the 128 bit wide XMM registers, and AVX on the 256 bit wide YMM registers. For each instruction set different extensions were implemented. The most recent are SSE4.1 and AVX2.[5]

[3]http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, last visited 10.2.2015

[4]http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html, last visited 10.2.2015

[5]https://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-

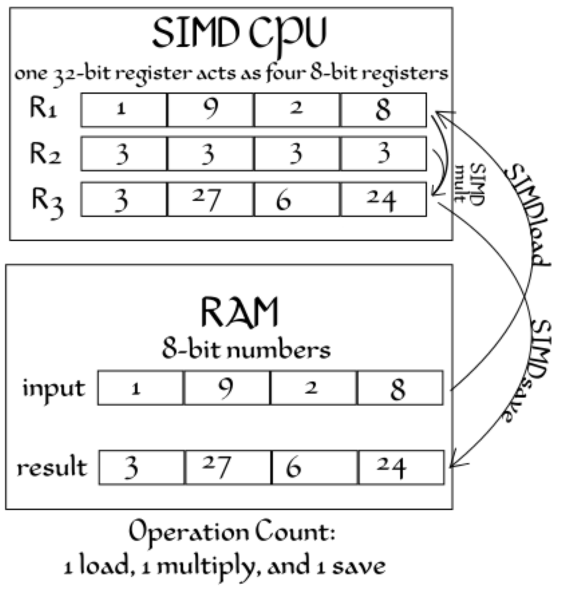


Figure 2.3: The illustration of multiplying 4 numbers with 3 using SIMD operations. The CPU loads 4 integers at once, multiplies them all in one SIMD-multiplication, and stores them all at once back to the result vector. In theory, the speed up is about 75%.
Source: "SIMD cpu diagram1" by Decora at en.wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - https://commons.wikimedia.org/wiki/File:SIMD_cpu_diagram1.svg

Intel intrinsic instructions are C-style functions, which provide access to the instructions implemented in the SIMD instruction sets. These functions wrap the calls to the SIMD instructions to prevent directly writing assembly.

2.3.4 Parallelising sequence alignments on CPUs

The challenge with parallelising dynamic programming algorithms like the Needleman-Wunsch and Smith-Waterman is that the calculations are inter-dependent. Each cell of an alignment matrix is calculated based on the values of its preceding cells (see section 2.2.3). Figure 2.4 shows different approaches of parallelising sequence alignments, by grouping multiple cell calculations in a SIMD vector.

The approach (A), by Wozniak [1997], in figure 2.4, describes a vectorisation along the anti diagonal. Here the values of the cells do not depend on values calculated in the same vector. Although loading the data gets more complex due to the loading of non-consecutive data. The second approach (B) was described in 2000 by Rognes and Seeberg [Rognes and Seeberg,

intel-integrated-performance-primitives-to-accelerate-algorithms, last visited 10.2.2015

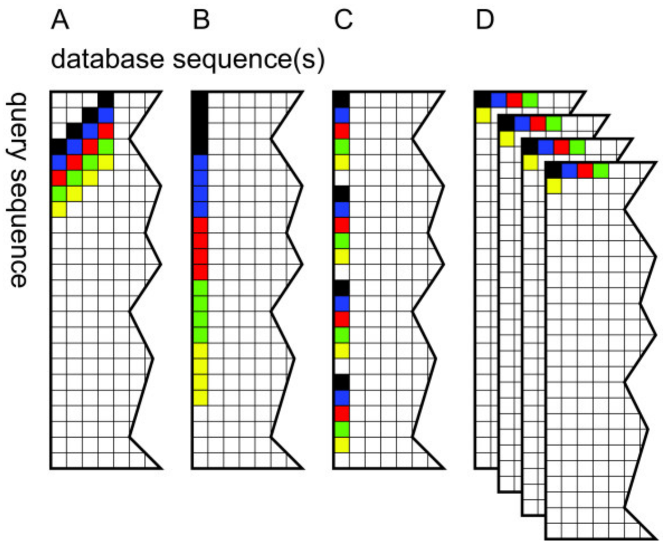


Figure 2.4: Different approaches to the vectorisation of sequence alignments. Alignment matrices are shown with the elements that form the first five vectors processed indicated in black, blue, red, green and yellow. For simplicity, vectors of only 4 elements are shown, while normally more elements would be used. Source: [Rognes, 2011]

2000]. It implements a sequence alignment along the query sequence, taking into account that cells in one vector depend on previous cells in the same vector. For local alignments, computed with the Smith-Waterman, it has been shown that this data dependency in a vector is the exception. To further reduce the data dependency, Farrar [2007] proposed approach (C). He shows that his striped approach reduces the data dependency even more and speeds up the alignments. In 2011 Rognes proposed another approach (D), computing multiple sequence alignments in parallel, instead of parallelising one sequence alignment [Rognes, 2011].

All the different approaches shown in figure 2.4 were proposed and implemented for local alignments using the Smith-Waterman algorithm with Gotoh's modification. Approach (A) and (D) can be applied to global alignments without further adjustments. (B) and (C) cannot be applied to global alignments without checking the data dependencies first. The difference here is that local alignments do not take gaps into account, at the beginning and the end of the sequences. This could be the reason for the reduced data dependency along the vertical vectors.

2.4 API design in C

Sequence alignments are used in variety of applications (see section 2.2.1). To speed up the development of an application it is common practise to implement the functionality for the different features, like sequence alignments, in libraries. This modular approach allows to use once implemented features in more than one application. The Application Programmable Interface (API) specifies the public functions of the library, and how it can be integrated in an application. It provides access to the functionality while hiding its implementation details. Interfaces describe a contract between the clients of an API and the implementation. The implementation provides the functionality, and the clients must use it in accordance to the rules described in the API. [Hanson, 1996, p. 24]

2.4.1 General design rules

A well designed API follows some general design rules:

- Keep it simple, stupid
  This is a universal principle and applies not only to software development. It is the base for the following design rules and encourages to create a clear and easily understandable API.[6]

- Keep it consistent
  Well designed APIs are consistent in their implementation. If a user understands how to call one function of the API, he immediately knows how to call the other functions. This includes having the same structure for all functions, as well as using only one or a limited set of design patterns. The structure of functions includes, among others naming, how to pass arguments or return values, and the ordering of parameters.[7]

- Do not make the client do, what the API can do
  If every client has to prepare parameters in the same way before providing them to the API, the API might as well prepare these parameters itself. This helps to reduce boiler plate code, and thus to improve the code calling the API.[8]

- Provide a good documentation
  A well designed API does not need a lot of documentation, its provided functions speak for itself. The documentation provides a fast entry point into using the API. This includes sample code, showing example requests and responses for calling the functions, and explanations of the error handling.[9]

- Provide a test-suite
  A provided test-suite can check, if a library runs correctly on a system. It helps to check, if possible errors occur within the library or the calling application. Additionally it documents the usage of the API.[10]

[6]https://stackoverflow.com/questions/2619854/best-practices-and-guidelines-for-designing-an-api, last visited 2.4.2014

[7]https://programmers.stackexchange.com/questions/100792/api-design-pitfalls-in-c, last visited 2.4.2014

[8]http://lcsd05.cs.tamu.edu/slides/keynote.pdf, last visited 2.4.2014

[9]http://blog.programmableweb.com/2010/08/12/web-api-documentation-best-practices/, last visited 2.4.2014

[10]http://davidz25.blogspot.no/2011/07/writing-c-library-part-4.html, last visited 2.4.2014

2.4.2 Design rules for implementations in C

Every programming language provides its own linguistic mechanisms for implementing interfaces or APIs. Some languages have more advanced and some only limited support. The C programming language provides only one way of implementing APIs, and thus separating interfaces from the implementation: header files. A header file describes the macros, types, data structures, variables, and functions, usable by clients. These are implemented in `.c` files, which include the header file. This way an interface can have multiple implementations, for e.g. different platforms, while the implementation is hidden to the clients. [Hanson, 1996, p. 15 + 18]

Design pattern are a good way of keeping an interface simple and understandable. Every developer, who knows the pattern, understands the interface easier. One design pattern for C APIs is the "opaque pointer" pattern, also known as PIMPL ("pointer to implementation idiom"). This pattern describes, how to exchange data with the client using a struct, without exposing its actual implementation. The header file provides only the declaration of the struct (see listing 2.1). Hence the client has no knowledge about the struct, regarding its size or members. The API provides accessor-functions to the members of the struct. The actual implementation then is located in the `.c` file, implementing the functions provided by the API (see listing 2.2). [11][12][13]

Listing 2.1: Example for an opaque pointer header file

```c
/* obj.h */
struct obj;

/*
 * The compiler considers struct obj an incomplete type.
 * Incomplete types can be used in declarations.
 */
size_t obj_size( void );

int obj_setid( struct obj *, int );

int obj_getid( struct obj *, int * );
```

Listing 2.2: Example for an opaque pointer c file

```c
/* obj.c */

#include "obj.h"

struct obj {
    int id;
};

/*
```

[11]http://www.gotw.ca/gotw/028.htm, last visited 3.4.2014

[12]http://www.qnx.com/developers/articles/article_302_2.html, last visited 3.4.2014

[13]http://www.drdobbs.com/cpp/making-pimpl-easy/205918714?pgno=5, last visited 3.4.2014

```
 * The caller will handle allocation.
 * Provide the required information only
 */
size_t obj_size( void ) {
    return sizeof( struct obj );
}

int obj_setid( struct obj *o, int i ) {
    if( o == NULL ) return -1;
    o->id = i;
    return 0;
}

int obj_getid( struct obj *o, int *i ) {
    if( o == NULL || i == NULL ) return -1;
    *i = o->id;
    return 0;
}
```

2.4.3 Existing sequence alignment APIs

Optimal sequence alignment algorithms have existed since the 1970s. The Needleman-Wunsch algorithm for example was first described in 1970 [Needleman and Wunsch, 1970], and the Smith-Waterman in 1981 [Smith and Waterman, 1981]. In the past a lot of different libraries where developed implementing optimal alignments. Two of the more recent libraries, implemented in C and C++, are SSW and Opal.

SSW: SSW [Zhao et al., 2013] implements the Smith-Waterman algorithm using Farrars approach [Farrar, 2007]: the algorithm is running in parallel on multiple cores using SIMD instructions. The library is implemented in C under the MIT license[14] and the the source code can be downloaded from GitHub[15]. The API of SSW contains four functions:

- `ssw_init`
  Reads in the query sequence and creates a profile for it. The profile is returned as a pointer to a struct.
- `init_destroy`
  Frees the memory allocated for the profile struct.
- `ssw_align`
  Aligns a sequence to a reference sequence using the profile by means of the Smith-Waterman algorithm. The alignment can be configured via parameters. It returns the alignment as a pointer to a struct.
- `align_destroy`
  Frees the memory allocated for the alignment.

[14]http://opensource.org/licenses/MIT, last visited 20.4.2015
[15]https://github.com/mengyao/Complete-Striped-Smith-Waterman-Library, last visited 20.4.2015

Additionally the API provides two structs:

- `s_profile`
  Contains data for the profile of the query sequence.
- `s_align`
  Contains the alignment information of both sequences.

The struct `s_profile` is only used internally during the calculation of the alignment. Hence it is implemented using the opaque pointer pattern. The memory for both structs is allocated internally, and the API provides functions to free it.

Opal: Opal implements optimal sequence alignments based on Rognes' approach [Rognes, 2011]. It implements the Smith-Waterman, the Needleman-Wunsch, and two modes of semi-global alignments, based on the Needleman-Wunsch algorithm. The difference to Rognes implementation is the support of AVX2 instructions using 256 bit wide registers to compute twice as many sequences in parallel. It is developed in C++ under the MIT license with the source code being available on GitHub[16]. The API of Opal contains five functions:

- `opalInitSearchResult`
  Allocates and initialises the struct `OpalSearchResult`, which is passed to other functions.
- `opalSearchResultIsEmpty`
  Returns the value of `scoreSet` of `OpalSearchResult`.
- `opalSearchResultSetScore`
  Sets the value of `score` in `OpalSearchResult`.
- `opalSearchDatabase`
  Searches a database for a query sequence using one of the three global alignment modes.
- `opalSearchDatabaseCharSW`
  Searches a database for a query sequence using local alignments.

Additionally the API provides one struct:

- `OpalSearchResult`
  Contains information about the score and alignment of the query sequence with one database sequence.

In addition to the functions and structs, the API provides a couple of constants controlling the behaviour of the function.

The development of Opal started in December 2013, at about the same time as the development of libssa . When we became aware of Opal we

[16]https://github.com/Martinsos/opal, last visited 20.4.2015

decided to continue the development of libssa. The reasons were that Opal implements less features, than were planed for libssa, and it implements Rognes' approach a bit different than libssa. Chapter 5.1.6 gives a more detailed overview of the differences between Opal and libssa and discusses the performance of both.

2.5 Measuring performance

Computer programs can be compared in various ways by comparing the usability, the provided features, or the performance. For sequence alignments and database searches the runtime performance is most important. Here one wants to compare the runtimes of different alignment programs and configurations to find the best.

2.5.1 Measuring speed

The runtime performance of software can be measured either as the time, a program needs to complete, or the throughput. The number of tasks the program is completing during a certain amount of time. [Hennessy and Patterson, 2012, p. 48 ff]

The speed of a program is first of all the number of CPU clock ticks it needs to run. This number can converted to CPU time, using equation 2.8. This measures only the time the CPU is operating which excludes the time it is waiting on other components, like input or output devices (I/O). [Hennessy and Patterson, 2012, p. 49]

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \tag{2.8}$$

The problem of CPU time is that it is an isolated measure, ignoring other factors affecting the actual response time. The response time, or wall clock time, is the elapsed time a program needs to run a query. This is the time a user of a program will notice. It includes all input/output activities, disc and memory accesses, and any operating system overhead.

Another measurement of speed is throughput. This number defines how many tasks can be completed in a given amount of time. In the case of optimal sequence alignments throughput is often measured in GCUPS (giga cell updates per second). This number shows how many times a billion cells of a sequence alignment matrix can be calculated. Rognes, among others, uses this measurement [Rognes, 2011].

2.5.2 Amdahl's Law

Amdahl's law defines a metric to compute the effect of an optimisation on the overall performance. It shows that the performance gain is limited by the fraction of time the improved code can be used. [Hill and Marty, 2008]

Equation 2.9 shows the computation of the overall speedup by improving a fraction f of a computation by a speedup S.

$$\text{Speedup}_{enhanced}\left(f, S\right) = \frac{1}{(1-f) + \frac{f}{S}} \tag{2.9}$$

This equation serves as a guide, to how much speedup an improvement of the code, will give. It gives an idea, where to focus on, when trying to improve the speed of a program.

2.6 Testing

Tests are an integral part of software development. They reduce the risk of software failures and check if the software meets the requirements. Testing should be introduced in the software development process at an early stage. This way, the software can be designed for easy testing and defects can be found early before they get too complicated to fix.[17] [Martin, 2009, chapter 9]

Testing is done on different levels, of a software:

- Unit level
  Any function or module of a program that can be tested separately.
- Integration level
  Interfaces between components or interactions with other systems.
- System level
  The behaviour of the whole software as defined by the requirements.
- Acceptance level
  Is done by the customer using the software.

Each of these levels implements different types of tests:

- Functional testing
  Tests the behaviour of a system and what it should do.
- Non-functional testing
  Measures the characteristics of a software, e.g. response time, usability, etc.
- Structural testing
  Measures the thoroughness of testing through assessment of code coverage.
- Tests related to changes
  These are added when a defect has been detected and fixed. They confirm that the defect has been removed.

[17] http://www.istqb.de/downloads/finish/16/15.html, last visited 15.2.2015

For the development of a library, tests at unit, integration, and system level should be implemented. The tests at acceptance level, are later on performed by developers integrating this library into their software. The tests at unit level should cover all implemented functions and the features they include. The tests at integration level probe the connections between modules, like the integration of a database into the library. The tests at system level test the public functions provided in the API. These test the library from the users point of view.

For all these tests, the code coverage is measured to find untested functions and statements. Additionally, by measuring the code coverage, one can find implemented but unused source code. This is done by measuring the code coverage for the system tests and looking at the source code, annotated with coverage markers. Using C and GCC as a compiler, this is done using gcov and lcov. Gcov is a code coverage analysis tool, integrated in GCC, which is enabled by the compiler flag `--coverage`. Lcov generates a html report from the gcov analysis data showing the percentage of executed lines per file and function.

Non-functional tests should be implemented, if the requirements specify these, e.g. the maximum runtime or amount of used memory. These can also be used to find bottlenecks and starting points for optimisations. Chapter 2.5 describes measurements for these.

Chapter 3

Design

This chapter illustrates the design idea for the implementation of the sequence alignment library. The actual implementation details are described in the next chapter.

This first part, of this chapter section 3.1, describes the use cases and the functions of the API. It shows how the library is integrated in other programs and how databases are integrated into this library. The next part, section 3.2, describes the modular design of the library and how it can be extended to, for example, support more sequence alignment algorithms. Section 3.3 then describes the core part of the library, the parallelised sequence alignments.

3.1 Design of the API

The main purpose of an API is to document the publicly available functions of a library. It should show only, what is needed to use the library. The API of libssa takes a similar approach as the APIs described in chapter 2.4.3, while extending it, to offer a greater flexibility.

3.1.1 Use cases

To guide the design process of the API, a couple of use cases have been defined:

- Database searches using global alignments
- Database searches using local alignments
- Processing sequences in FASTA format[1]
- Support for nucleotide and amino acid sequences
- Support for translating nucleotide sequences to amino acid sequences, using different genetic codes[2]

[1]http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml, last visited 23.4.2015
[2]http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi, last visited 23.4.2015

- Results as scores only
- Results as scores and CIGAR strings

In addition to the functional requirements defined by the use cases a couple of non function requirements were defined.

One goal of the library is to evaluate the improvement of database searches through parallelisation. For this purpose, a number of non-functional requirements were defined, in addition to the functional requirements defined by the use cases. The non-functional requirements are all related to the database searches. These should run on multiple threads and they should be vectorised using both SSE and AVX instructions. The vectorised algorithms should be implemented in two versions, using 8 and 16 bit wide integers for the computation of the alignment scores.

3.1.2 Alignment API

The API is based on the functional and non-functional requirements, described earlier. The main idea behind the API is to have functions, initialising the library and functions, executing the database searches and alignments. This way, the initialisation is only done once and afterwards multiple searches can be performed without a constant reinitialisation. Another advantage of this design is a reduced amount of parameters per function. Each function is responsible only for one task like, for example, the initialisation of the score matrix.

Some functions carry a flag as a parameter, defining how to interpret other parameters. This is done as a trade-off between having too many functions and grouping functions of the same task. In the following, lists of all functions of the API grouped by their main task with a short description and the flags, are provided.

The first group of functions initialises the library. They set parameters, that are usually used for multiple different searches like the database, which does not need to be initialised for each search, but once in the beginning:

- `init_score_matrix`
  Initialises a score matrix.
  Flag: `mode` - defines if the second parameter is a file name, a matrix as a string, or a constant for a build-in matrix
- `init_constant_scores`
  Initialises constant match and mismatch scores to use instead of a score matrix.
- `init_gap_penalties`
  Initialises the gap penalties.

- `init_symbol_translation`
  Initialises the translation of symbols, if the query or database is translated to a protein sequence.

- `init_db`
  Initialises the external database.

- `init_sequence_fasta`
  Initialises a FASTA formatted query sequence. Flag: `mode` - defines if the second parameter is a file name or a FASTA formatted sequence as a string

The second group of functions performs the database searches, using an optimal alignment algorithm:

- `sw_align`
  Aligns a query sequence to a database using the Smith-Waterman algorithm.

- `nw_align`
  Aligns a query sequence to a database using the Needleman-Wunsch algorithm.

The last group of functions does not affect the alignment results. This group contains functions for deallocating memory used for storing alignment results and input data. Furthermore it contains functions to change the technical behaviour of the library, like turning on error reporting or setting the maximum number of used threads:

- `set_simd_compute_mode`
  Sets the maximally used SIMD capabilities. This can restrict the library to use the SSE implementation even if the AVX2 instruction set is available.

- `set_output_mode`
  Sets the output mode. This can either be stdout or no output.

- `set_chunk_size`
  Sets the number of sequences in a database chunk.

- `set_thread_count`
  Sets the number of threads, used by the alignment functions.

- `free_sequence`
  Deallocates memory used for a query sequence.

- `free_alignment`
  Deallocates memory used for storing alignment results.

- `ssa_exit`
  Terminates the library and database and deallocates all used memory.

In addition to these functions, the API provides two data structures for initialised data and the results of the sequence alignments. The first structure stores the query sequence profile and the second structure stores a list of the highest scoring sequence alignments. The data for each alignment contains both sequences, the score, and the CIGAR string and position of the computed alignment.

3.1.3 Database API

The library provides a second API, which is used to integrate external databases into libssa. This second API is a C header file which is implemented by database libraries to use these with libssa:

- `ssa_db_init`
  Initialises the database.
- `ssa_db_close`
  Closes the database connection.
- `ssa_db_get_sequence_count`
  Returns the number of sequences in the database.
- `ssa_db_get_sequence`
  Returns a sequence from the database.

Database sequences are returned from the database through a data structure, defined in the database API. This structure contains the sequence as an ASCII string, its length, and the database ID. The ID is a unique identifier of the sequence in the database.

3.2 Design of the library

The structure of the library was designed, for an incremental build model. It is composed of different modules, which are implemented and tested independently. This way, new algorithms or different implementations of one algorithm can be integrated easily. In addition to that, it supports loose coupling of components, like the database, which is not tied to this library and exchangeable. This simplifies the integration of libssa, into programs which already implement a database for sequences.

Figure 3.1 gives an overview of the implemented modules and their connections. Each colour shows a different level of modules, where each module is only connected to its previous and next level to keep dependencies low. The modules shown in yellow are the public functions of the API, as described in chapter 3.1.2. The modules shown in orange implement the actual alignment functions with different characteristics. The modules in between call the alignment functions based on the parameters passed through the public API. The `manager` module initialises the `searcher` and `aligner` with the alignment function called from the public API. Afterwards it starts the `searcher` module, waits until it is finished and either calls the `aligner` module, for computing the alignments, or returns the results of the `searcher` module directly. Both run in multiple threads, managed by the `manager` module. During the initialisation, the `searcher` module chooses one of the search functions, based on the set SIMD capabilities and bit width. Each search function has a fall-back to the next higher bit width, if the alignment score exceeds the current bit width. The search functions return the highest scoring database sequences including the scores. The `aligner` module takes the result of the search functions and computes a CIGAR string, representing the actual alignment.

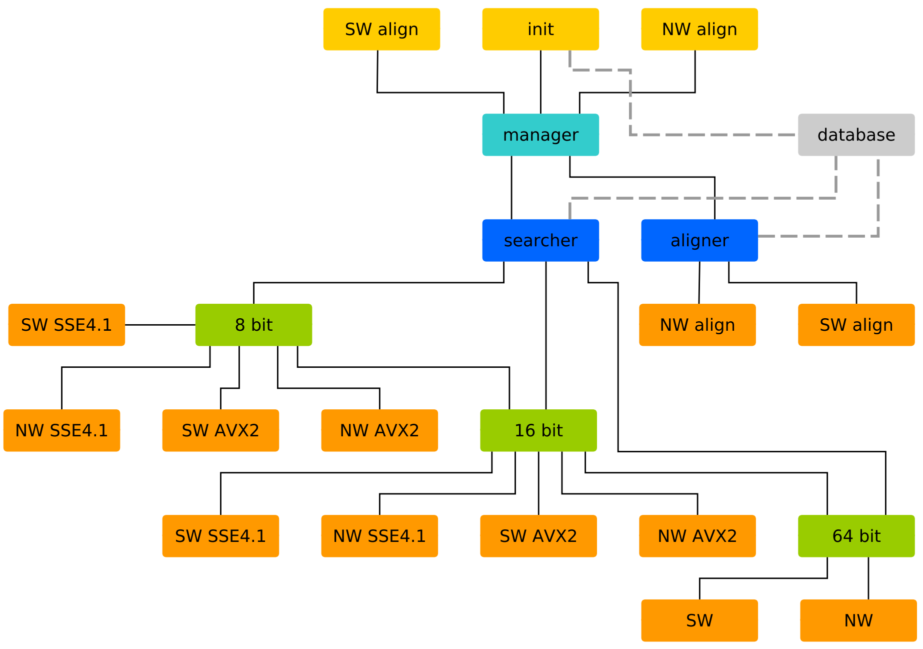


Figure 3.1: The graph shows the modular design of libssa. Each colour shows a different level of abstraction. The API functions, shown in yellow, make requests to the `manager` module. This delegates the requests to the `searcher` and `aligner` modules, which themselves delegate these further down to the database search and alignment algorithms. The database, shown in grey, is integrated as an external library.

The database, shown in grey, is integrated through a second API (see chapter 3.1.3). This API has to be implemented by a database library, that is used with libssa. The database library is then linked to libssa during compile time. Figure 3.2 shows the integration of a database into libssa. The database is initialised through a function of the public API, while the `db_adapter` module encapsulates the database accesses and prepares the database sequences. The `searcher` and `aligner` modules call the `db_adapter` only, without having any knowledge about the database.

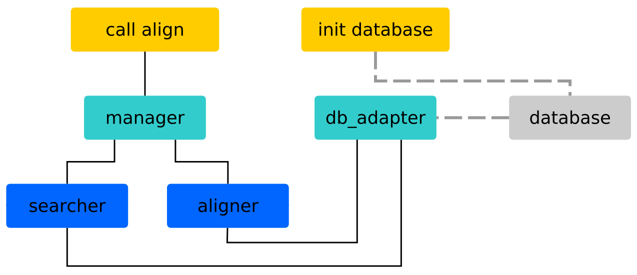


Figure 3.2: The graph shows the integration of an external database into libssa. Each colour shows a different level of abstraction. The database is here initialised through an API function. The `db_adapter` module wraps all requests to the external database library.

3.3 Parallelisation of sequence alignment

The main focus of the library lies on fast optimal sequence alignments. Libssa implements these following Rognes' approach, as described in chapter 2.3.4: multiple sequence alignments are computed in parallel. The parallelism here is implemented on two levels. The first level distributes the database sequences to multiple threads while the second level aligns the query sequence against multiple database sequences in parallel, using the SIMD capabilities of the CPU.

This is the same approach to parallelism as proposed by Rognes and as implemented in SWIPE[3] and VSEARCH[4]. The key difference in libssa is the modularisation. Libssa is developed to be extend with new algorithms and improved implementations of the current algorithms, while SWIPE and VSEARCH are implemented with only one algorithm in mind.

3.3.1 Threads

On the first level of parallelisation, the workload is distributed to multiple threads. This parallelisation is implemented in a separate module and applied to database searches and the computation of alignments. Hence the workload is either the database sequences, a query sequence is searched in, or the pairs of database and query sequence, for which an alignment is computed. Figure 3.3 outlines the implementation design for both cases. First the threads are initialised with data common to all. This data contains among others the query sequence and the used algorithm and bit width. Afterwards a number of threads is started and the `manager` module waits for their termination. Each thread processes its workload in chunks until an empty chunk is reached which signals that no work is left. For each chunk, the configured algorithm, shown in green, is called. Upon termination, each thread returns a result list. In the `manager` module, these are combined

[3]https://github.com/torognes/swipe, last visited 15.4.2015
[4]https://github.com/torognes/vsearch/, last visited 15.4.2015

into a global result list. Each thread produces as many results as returned in the global result list while the global result list contains the best results of all thread results.

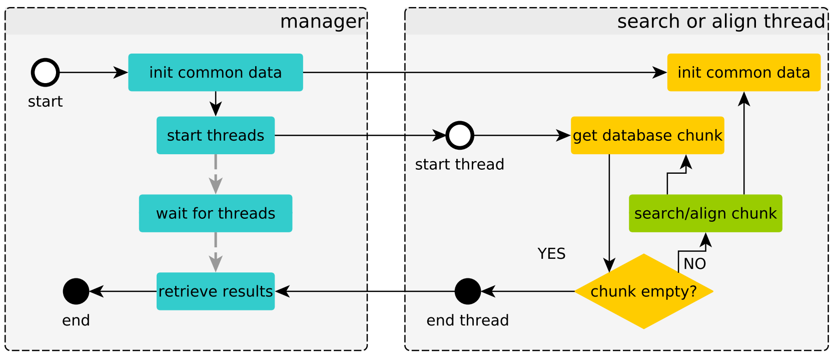


Figure 3.3: The chart shows the data flow and the communications between the `searcher` and `aligner` threads and the `manager` module. First the threads are initialised with some common data. Afterwards these are started and each thread processes chunks of data, based on the common data. Meanwhile the `manager` module waits until the threads are finished and collects their results upon termination.

3.3.2 Non vectorised sequence alignments

One task of libssa is to asses the performance improvements of computing sequence alignments using SIMD vectors. Therefore a non vectorised version of the Smith-Waterman and Needleman-Wunsch algorithm is added to libssa, to perform as a baseline for evaluating the performance of the vectorised database searches. The non vectorised searches are executed on the first level of parallelisation, on multiple threads.

Additionally do they perform as fallbacks, if the vectorised versions cannot be used. This can be the case if the used computer does not implement the required SIMD capabilities.

3.3.3 Vectorised alignments

The second level of parallelisation is realised using SIMD instructions (see chapter 2.3.3). This is done by splitting up the SIMD vectors into multiple channels. Each channel computes the alignment of one database sequence and the query sequence. The computations are done in parallel. Here, libssa implements Rognes' approach for vectorising sequence alignments from 2011 [Rognes, 2011]: one query sequence is aligned to multiple database sequences in parallel. The design of the implementation follows the implementations of Rognes' approach in SWIPE and VSEARCH.

Figure 3.4 shows the flow chart for aligning one query sequence to a chunk of database sequences. In the first iteration, all channels are empty and no sequence has ended. Each channel is filled with a sequence until all channels are filled, no sequence is left. The remaining channels are set to 0. In the next step, the search window over the loaded database sequences is set to the next block of four residues, of each channel (shown in green). This block is used to compute a temporary score profile (shown in blue) which is used to align the block to the query sequence (shown in orange).

After each alignment of a block, the algorithm checks if a sequence in a channel has ended. If so, the alignment score for that sequence is saved and a new sequence is loaded. Afterwards, the search window is moved to the next block, a new score profile is computed, and this block is aligned. If no sequence ended, the search window is moved to the next block, a new score profile is computed, and this block is aligned.

Splitting the algorithm in two states reduces the amount of instructions executed during the alignment. The simple state, executed when no sequences have ended, simply moves the search window to the next block, computes a new score profile, and aligns the block to the query. Whereas the complex state additionally saves the alignment scores of the finished sequences, loads new sequences, and initialises the alignment of the new sequences.

Figure 3.5 shows the database sequences, lain out in eight channels and blocks of four symbols. Each block resembles one state of the database search window. The vectors shown in yellow, orange, green and blue are aligned in the first block in this order. The cells in red highlight the last residues of database sequences while cells containing a dash denote padding of database sequences. The padding is used to fill up sequences not ending at a block.

To speed up the computation, a temporary alignment profile is created. This profile contains the scores for aligning the database residues of one block to the query sequence. It allows for a fast loading of the scores into the vectors used for computing the alignments. Figure 3.6 illustrates the creation of this profile. The score matrix is reduced to the nucleotides A, C, G, and T, to simplify the illustration, and the block of database sequences shows only eight channels, where later more channels are used. The numbers in bold face mark the positions of the four residues and the colours mark the columns of the score matrix representing one residue and their positions in the score profile. The profile then consists of the columns, of the score matrix, representing the residues of the sequence block. Each column contains the scores for aligning the database residue to all possible query residues.

The alignment of the database sequences to the query sequence is done block wise. Each database residue of a block is aligned to each query

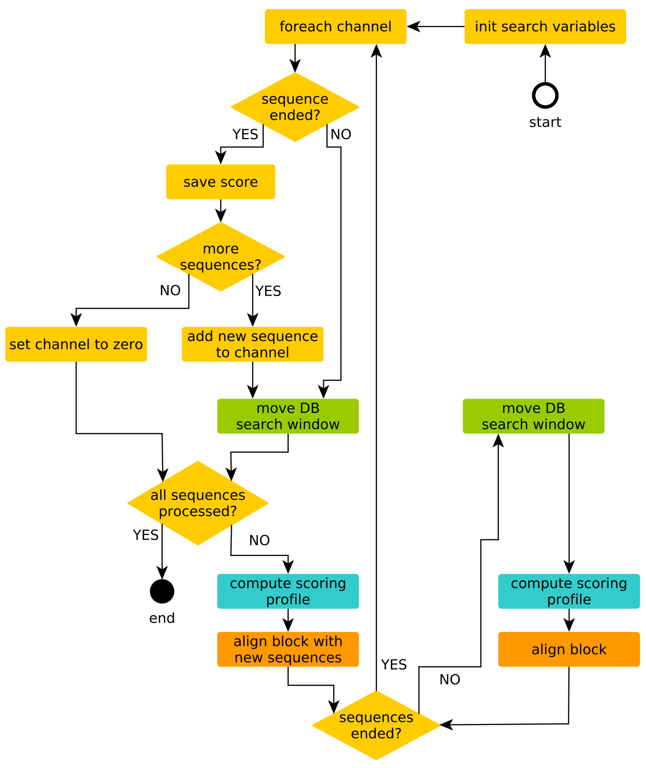
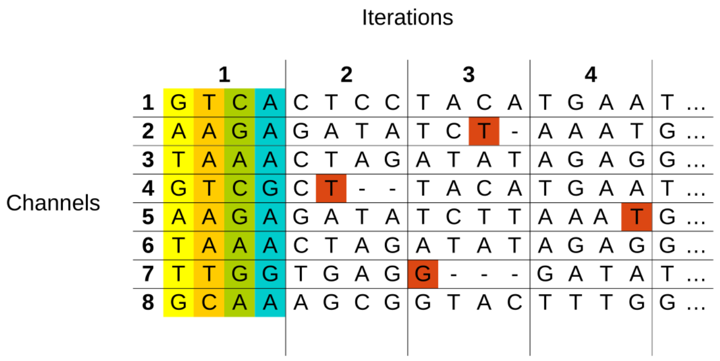


Figure 3.4: The flow chart shows a database search in one chunk. The processing is done in two states, while the actions shown in green, blue, and orange are common two both states.



	1				2				3				4				
1	G	T	C	A	C	T	C	C	T	A	C	A	T	G	A	A	T ...
2	A	A	G	A	G	A	T	A	T	C	T	-	A	A	A	T	G ...
3	T	A	A	A	C	T	A	G	A	T	A	T	A	G	A	G	G ...
4	G	T	C	G	C	T	-	-	T	A	C	A	T	G	A	A	T ...
5	A	A	G	A	G	A	T	A	T	C	T	T	A	A	A	T	G ...
6	T	A	A	A	C	T	A	G	A	T	A	T	A	G	A	G	G ...
7	T	T	G	G	T	G	A	G	G	-	-	-	G	A	T	A	T ...
8	G	C	A	A	A	G	C	G	G	T	A	C	T	T	T	G	G ...

Figure 3.5: Layout of the database sequences by channels and blocks. Cells in red indicate the last residues of sequences and dashes indicate padding. The columns shown in yellow, orange, green, and blue show the order of processing, in one search block. The yellow column is processed in the first vector.

residue, before loading the next block. Here the formulas for global and local alignments with the Gotoh extension, as described in chapter 2.2.3, are applied. The vectors of substitution scores from the score profile match here to the value V in the Gotoh formula in chapter 2.2.3.

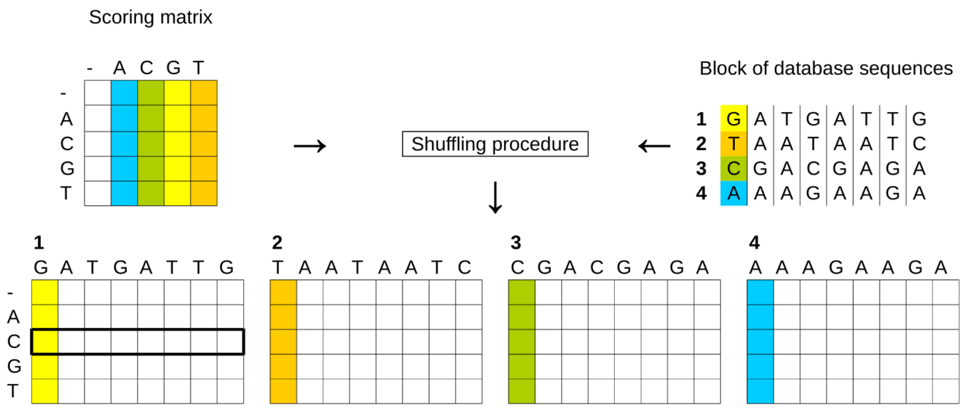


Figure 3.6: Creation of the database profile for a block of four residues of eight database sequences using a reduced score matrix. The columns of the score matrix are selected and placed in the score profile, based on the residues in the search block. The vector, marked with a bold border, denotes the vector, that would be loaded to align the query residue C to the first vector of database residues.

An important part of the implementation over multiple channels is the initialisation of new sequences. Since every value is kept in a SIMD vector, one has to make sure that the values for the H, E, and F matrices are initialised according to the algorithms. This requires access to the separate channels to initialise only the affected ones. The same goes for the score of the alignments. Here only the score of the terminated sequences is read and reset, while the rest of the scores are unaffected.

Chapter 4

Implementation

This chapter describes the implementation details of the sequence alignment library. The first part describes the different configuration functions and the internal data formats. The second part introduces the integration of an external database into libssa, followed by details of the central module. This module initialises and starts database searches and alignments. The next part, section 4.4, describes the implementation of the database searches, followed by details of the computation of alignments. The final two parts, present testing of libssa and benchmarking.

Some parts of the implementation were taken from other open source projects. These parts are mentioned here, while a complete discussion is given in chapter 5.3.

4.1 Configuration and internal formats

This section focuses on the configuration steps and how the configured data is stored internally. Emphasis is on the API functions, for initialising the library. The database search and alignment functions are described in the following chapters.

4.1.1 Data types

In libssa a couple of C data types were chosen as a convention for certain types of data. This was done to unify the code and to reduce errors. Besides this, libssa declares a number of C structs, for storing and exchanging data. Internal data types are declared in the header file `libssa_datatypes.h`, while external data types are declared in the header file `libssa.h`.

In libssa, all sequence data is stored in arrays of the type `char`. The length of the sequences is stored in a variable of type `size_t`. This is the standard data type, for the size of an object or sequence, as defined by the C99 ISO/IEC 9899 standard[1]. It might be different on each system, but

[1]http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57853, last visited 4.4.2015

it is defined to be big enough, to hold the maximum size of any object, allocatable on that system. In libssa `size_t` is used for the size of all arrays and objects. Alignment scores are always stored in variables of type `signed long`, when returned to the user. Although internally they might be stored in smaller data types, depending on the algorithm.

4.1.2 Sequences

Libssa uses FASTA[2] as input format for query sequences and ASCII strings for database sequences. In both the sequence data is represented in the standard IUB/IUPAC amino acid and nucleic acid codes[3]:

- 17 Nucleotides: `-ACGTUMRWSYKVHDBN`
- 28 Amino acids: `-ARNDCQEGHILKMFPSTWYVBZX`

Internally, these sequences are represented as sequences of integers, in the range of 0 to 15 for nucleic acids and 0 to 28 for amino acids. Here libssa does not distinguish between the nucleotides U (uracil) and T (thymine), as both cannot occur in the same sequence. Therefore the internal format only uses 16 different nucleotide codes. The translation to internal codes is done to directly use them later as indices in the score matrix.

Unknown symbols are set to 0 – the representation of the gap symbol – during the mapping to the internal format. If unknown symbols are found a warning is shown.

Query sequence The query sequence is read in FASTA format and stored as a sequence profile. It is then read from a file or a string. The file allows for a header to be present, while the string only contains the sequence leaving the header field of the profile empty. The computed profile contains the sequence in its internal representation in the selected translation mode, the length of the sequence, the header string, and the length of the header string. Only the sequences used for the selected mode are stored.

The query profile is implemented in the opaque pointer pattern, allowing for storage of the profile on the user side while hiding the implementation.

Symbol translation Libssa implements 5 different modes for aligning sequences where the query and database sequences are translated, if necessary (see chapter 2.2.2). It is, for example, possible, to search for a nucleotide sequence in a protein database and vice versa. To configure this, the symbol type, the used strands, and the genetic codes of the query sequence and the database are set. The following list shows the implemented modes and the constants (in the file `libssa.h`) for selecting these:

[2] http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml, last visited 23.4.2015

[3] http://pac.iupac.org/publications/pac/pdf/1972/pdf/3104x0639.pdf, last visited 23.4.2015

- `NUCLEOTIDE`
  Aligns a nucleotide query against a nucleotide database.
- `AMINOACID`
  Aligns an amino acid query against an amino acid database.
- `TRANS_QUERY`
  Translates a nucleotide query and aligns it to an amino acid database.
- `TRANS_DB`
  Aligns an amino acid query to a translated nucleotide database.
- `TRANS_BOTH`
  Translates a nucleotide query and aligns it to a translated nucleotide database.

The translation is done on the internal integer representation. It is either done for the provided strand, the complementary strand, or for both strands. For all strands, all 3 reading frames are translated, resulting in up to 6 translations of a sequence.

For translations, libssa implements the list of genetic codes, provided by the NCBI[4]. This list contains genetic codes, for 19 different groups of biological organisms and organelles. In libssa the codes are identified by the numbers used at the NCBI, while different codes can be used for translating the query and database sequences. The implementation of translations was taken from the SWIPE project[5] and adapted to libssa.

4.1.3 Scoring schemes

Libssa implements two scoring schemes: constant match and mismatch scores, and substitution matrices. Constant scores are passed to the library through the function `init_constant_scoring` in the range of -128 to $+127$. Substitution matrices are passed to the library using the function `init_score_matrix` as a file or string, or by selecting one of the implemented matrices: BLOSUM45, BLOSUM50, BLOSUM62, BLOSUM80, BLOSUM90, PAM30, PAM70, or PAM250.

Listing 4.1 shows an extract from the BLOSUM90 matrix in the format readable by libssa. Lines starting in # and blank lines are ignored. Lines starting in a space or tab character set the order of symbols (line 2 of the listing), and the rest of the lines set the values. Matrices up to 32 x 32 symbols with scores in the range of -128 to $+127$ are supported. Internally, the symbols are translated into the internal representation where each symbol has a value equal to its index in the score matrix. Therefore the order of symbols provided by a user is not important to libssa.

[4]http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi, last visited 12.4.2015
[5]https://github.com/torognes/swipe, last visited 10.4.2015

Listing 4.1: Part of the BLOSUM90 substitution matrix, in a format that is read by libssa.

```
# Entries for the BLOSUM90 matrix at a scale of ln(2)/2.0.
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F
A  5 -2 -2 -3 -1 -1 -1  0 -2 -2 -2 -1 -2 -3 ...
R -2  6 -1 -3 -5  1 -1 -3  0 -4 -3  2 -2 -4
...
```

Internally, the library uses only substitution matrices. Constant scores are converted into matrices. For each search range (8, 16, and 64 bit) a separate matrix is allocated and initialised. This avoids conversions between variables in different bit width during the computations.

4.1.4 Gap penalties

Libssa implements linear and affine gap costs, in the range of -128 to $+127$. Both are initialised through the same function (`init_gap_penalties`) to keep the API simple. The function takes two signed 8 bit integers as parameters: one for the gap opening and one for the gap extension costs.

Internally the gap costs are applied using the formula $C_{gap} = Q + n * R$, with C_{gap} as the costs for a gap of length n, and Q as the gap opening and R as the gap extension costs. If Q is set to 0, this formula calculates linear gap costs instead of affine gap costs.

The range of -128 to $+127$ was chosen, for two reasons: usually gap costs are much lower and secondly using gap costs of a higher range would make 8 bit searches impossible, as these would overflow at the first gap.

4.1.5 Validation of the configuration

Database searches are executed in two phases in libssa. First a search is initialised and secondly the search is performed. Each module validates the configuration up front to prevent errors due to a wrong or missing initialisation.

The concept here is to validate the configuration as early as possible to reduce the amount of testing in modules further down the hierarchy, since they are more likely to be executed more often. Hence performance is improved by moving the validation upwards. Each module implements a test on the data that is set during the initialisation. If the data is not present, an error is thrown and the computation is terminated.

Another advantage of early validation is a reduction in the amount of code testing for errors. C does not have a simple exception handling strategy and adding an error return value or parameter to many functions, bloats up the code.

4.2 Database integration

Libssa is designed to work with any database implementation, that implements a header file provided by libssa: `libssa_extern_db.h`. The functions of this header file are called by the `db_adapter` module to load and prepare database sequences. The prepared sequences are then used in the database searches and alignments.

The struct `seqinfo` (see listing 4.2) represents the minimum amount of data needed, to perform a database search or alignment. `ID` is a unique identifier used to identify the sequence in the database. Libssa shows this ID in the results of a search. `Seqlen` contains the length of the sequence and `seq` contains the sequence as an ASCII string without the header (see chapter 4.1.2).

Listing 4.2: Data struct for exchanging database sequences

```
struct seqinfo {
    size_t ID;
    size_t seqlen;
    char * seq;
};
```

Libsdb is a reference implementation, of an external database library. It is available on GitHub[6] under the AGPL v3 license[7]. It implements the minimal features, required by libssa, as defined in `libssa_extern_db.h`. The databases are read in FASTA format and stored in memory. The `ID` of a sequence is implemented as the index of the sequence, in the database file, starting from zero.

4.3 Controlling database searches and alignments

The central module controlling the database searches and alignments is the `manager` module. Its task is to start the database searches and alignments on multiple threads and to collect the results afterwards.

The module works in two phases, like all modules of libssa. First it is initialised by the calling module and initialises the modules it is going to call, secondly it performs the actual database searches and alignments. The initialisation is done using the functions `init_for_sw` and `init_for_nw`, depending on the used algorithm. Afterwards a database search and alignment is started using the function `m_run`.

The variable `align_type` controls whether CIGAR strings are computed for the search results or only the alignment scores are returned. If it is set to `COMPUTE_SCORE`, the database search is done and only the alignment scores are returned. For the additional computation of CIGAR strings, it is set to

[6]https://github.com/RonnySoak/libsdb, last visited 5.4.2015

[7]http://www.gnu.org/licenses/agpl.html, last visited 5.4.2015

`COMPUTE_ALIGNMENT`. The flag is set via the alignment functions of the API which receives it as a parameter.

The `manager` module delegates the thread handling to a separate module implementing a thread pool. It does the same with the collection of results which is delegated to a min-max-heap. This way the actual handling of threads and results becomes transparent to the `manager` module which not only simplifies the implementation, it also makes it easy to exchange these.

4.3.1 Thread pool

A simple thread pool is implemented, using the `pthread` library for executing computations in parallel. The number of threads can be specified by the user through the API function `set_threads`. If it is not specified, a number of threads equal to the number of logical cores is used. Once created threads are re-used until the thread pool is terminated or the number of threads changes.

After the initialisation, the function `start_threads` is used to execute the threads. The function `wait_for_threads` is afterwards used to wait for the termination of the threads and to collect the results of each.

The `pthread` library requires the functions implementing the threads to have an argument of type `void *`. The return values of threads are collected in a variable of type `void **`. This allows for a simplification of the `pthread` API, but makes the user responsible of checking the parameters, since a `void` pointer can contain basically any data. In libssa, this is done by documenting the type of the parameter. This is sufficient, since this part is internally used only and not exposed to the user.

4.3.2 Min-max-heap

A priority queue, implemented as a min-max-heap [Atkinson et al., 1986], was chosen, for collecting the best sequences. The implementation was taken from the VSEARCH project, since it was already developed for time and memory efficiency.

The implementation was then adapted to libssa by changing the data stored in an element of the heap and a set of unit tests was added. Listing 4.3 shows the data structure for an element. This is the minimum amount of data needed to identify a query and database sequence combination. The database sequence is identified using the `db_id`, `dframe`, and `dstrand`. The information about the strand and frame is used, if the database is translated for the search, while `db_id` is the ID of a sequence as defined by the database library. The version of the query is identified by `query_id`, which identifies the used translation of the query. The field `score` stores the computed score of the alignment. The score is the information the values are sorted by.

Listing 4.3: Data struct for elements of the min-max-heap

```c
typedef struct {
    size_t db_id;
    uint8_t db_frame;
    uint8_t db_strand;
    uint8_t query_id;
    long score;
} elem_t;
```

The implemented version is a minimums heap keeping the element with the lowest score at the top. New elements are added at the right position and elements with a lower score are shifted towards the top. If the heap is full, the element at the top is removed. The size of the heap is configured by the parameter `hit_count` of the alignment functions.

Each thread maintains its own priority queue for collection the highest scoring sequence pairs. The `manager` module retrieves these priority queues, upon termination of the threads and combines them to a global result. The size of each priority queue is the same to ensure the collection of the best results, even if these are collected in only one thread. The global result list then keeps the best n sequences, where n is equal to the value of the parameter `hit_count`.

4.4 Database searches

Database searches compute the alignment scores and return the best matching sequence pairs. This part describes the implementation details of the searches, focusing on the 8 and 16 bit SIMD implementations, which follow the general design as described in chapter 3.3.3.

The database searches are implemented in a modular structure like shown in figure 3.1 on page 31. This structure implements the `searcher` module on the highest level, with specialised search modules, focused on different bit widths, on the next level. These again delegate the work to the different implementations of the sequence alignment algorithms. The modules for the different bit widths retrieve chunks of sequences from the database, pass these to the alignment algorithms, and collect the best scoring sequences in local min-max-heaps. The local results are then returned to the `manager` module through the `searcher` module.

All the modules performing the database searches are implemented as a plug-in structure using C function pointers. In the initialisation phase, each module selects the module on the next lower level based on the parameters `bit width` and `search type`. The bit width is either 8, 16, or 64 and the search type Smith-Waterman or Needleman-Wunsch. Using function pointers moves the decision which implementation to take into the initialisation phase, while generalising the source code for the search phase. The modules on one level are all implemented the same way. All public functions have the same signature, the actual implementation then is specialised.

The searches in different bit widths are implemented to use the SIMD vectors best. The challenge here is to pack as many computations as possible in a vector, which is achieved using 8 bit integers. The problem is, that these only have a range of 0 to 255 unsigned or -128 to $+127$ signed. This is easily exceeded, depending on the match and mismatch costs, the gap penalties, and the length of the sequences. Therefore the different bit widths are implemented, serving as fallbacks for computations exceeding a bit width:

- 8 bit SIMD (range: 0 to 255 or -128 to $+127$)
- 16 bit SIMD (range: 0 to 65 535 or $-32\,768$ to $+32\,767$)
- 64 bit serial (range: 0 to 2^{64} or -2^{63} to $+2^{63}-1$)

In the 8 and 16 bit versions over- and underflows are detected and the affected sequences are re-computed with the next higher bit width. An overflow using 64 bit integers is practically impossible, since the number of base pairs in the longest known genome (see chapter 2.2), is still more than 7 orders of magnitude lower than 2^{63}.

For each bit width two sequence alignment algorithms are implemented, the Needleman-Wunsch and the Smith-Waterman, doing global and local alignments. For the 8 and 16 bit SIMD implementations two versions of each algorithm are implemented, one using the SSE and one the AVX instruction set.

When a search is started, one bit width, one SIMD version, and one algorithm is selected. The database search is then started in this bit width. If necessary, overflowing sequences are re-computed in the next higher bit width with the same SIMD version and algorithm. The re-computation is simply done by delegating the computation to the search module with the next higher bit width. This was already initialised beforehand using the same SIMD version and algorithm.

4.4.1 64 bit implementation

The 64 bit versions of the local and global alignments compute the scores sequentially. For each chunk one score is computed after the other and then added to the min-max-heap. The Smith-Waterman implementation is taken from the SWIPE project, while the Needleman-Wunsch implementation is derived from the Needleman-Wunsch-Sellers implementation in SWARM. Both are adapted to libssa regarding the parameter lists. Neither is optimised with SIMD instructions, since they only serve as fallbacks if the 16 bit range is exceeded.

4.4.2 8 and 16 bit implementations

The implementations for 8 and 16 bit searches is based on the 16 bit Needleman-Wunsch implementation of VSEARCH. It implements Rognes' approach from 2011[Rognes, 2011] using SSE2 instructions and was taken as a reference to implement four versions (8/16 bit and SSE/AVX) of both algorithms, the Smith-Waterman and the Needleman-Wunsch.

All versions follow the same general procedure as described in chapter 3.3.3. This is possible, since all used integer vector operations, are available in the SSE and AVX instruction sets. The difference between the 8 and 16 bit implementations is the number of channels per vector and the available score range.

Source code organisation The source code for the vectorised database searches is located in the following files in the sub directories `8`, `16`, and `simd` of the source folder `algo`:

- `search_simd_nw.c`
  8 and 16 bit Needleman-Wunsch implementation, for SSE and AVX instruction sets.
- `search_simd_sw.c`
  8 and 16 bit Smith-Waterman implementation, for SSE and AVX instruction sets.
- `search_16_util.h` and `search_16_util.c`
  Utility functions and API for the 16 bit implementations.
- `search_8_util.h` and `search_8_util.c`
  Utility functions and API for the 8 bit implementations.

The main challenges, of this part of the library are to implement it as efficient as possible and with as little code repetitions. The efficient implementation is described further down, while the reduction of code repetitions is done here on file level.

The 8 and 16 bit versions for both instruction sets follow the same general procedure. Hence it is possible to implement them in one file using C macros while still keeping it readable. At compile time these are resolved with the actual types and functions of the selected version, which again are selected using conditional preprocessor macros. The macros for the conditionals are defined using compiler arguments. The argument `-DSEARCH_8_BIT` sets the macro `SEARCH_8_BIT` for choosing between the 8 and 16 bit versions, while the compiler flag `-mavx2` sets the macro `__AVX2__` for choosing between the AVX and SSE versions. The flag `-mavx2` tells the compiler to optimise for a CPU implementing the AVX2 instruction set.

Each of the files is compiled optimised for the minimally required instruction set. The idea here is to make the library usable, even if the CPU does not implement, for example, AVX instructions. In this case the SSE versions can still be used. The minimum requirements for the 16 bit SSE database searches is SSE2, for the 8 bit SSE searches SSE4.1, and for the 8 and 16 bit AVX searches it is AVX2.

Listing 4.4 shows an example of the preprocessor macros for setting the intrinsics function for a saturated addition. In the implementation the first parameter of the define (`_mmxxx_adds_epiYY`) is used, which is resolved by the preprocessor to the second parameter. All of the functions set like this, follow a common naming scheme: `_mm` marks a function as an intrinsics function, `xxx` shows that this function has an 128 bit (SSE) and 256 bit (AVX) version, and `YY` indicates that this function has an 8 and 16 bit version.

Listing 4.4: Preprocessor macros for setting the functions for the selected version.

```
#ifdef SEARCH_8_BIT
    #ifdef __AVX2__
#define _mmxxx_adds_epiYY _mm256_adds_epi8  // 8 bit AVX
    #else
#define _mmxxx_adds_epiYY _mm_adds_epi8     // 8 bit SSE
    #endif /* __AVX2__ */
#else
    #ifdef __AVX2__
#define _mmxxx_adds_epiYY _mm256_adds_epi16 // 16 bit AVX
    #else
#define _mmxxx_adds_epiYY _mm_adds_epi16    // 16 bit SSE
    #endif /* __AVX2__ */
#endif /* SEARCH_8_BIT */
```

Compilers are most effective at optimising loops when the number of iterations is constant and known at compile time. To support this, a couple of constants were added to the conditional macros. These set, for example, the number of channels, which is constant to one bit width and instruction set, but different for other configurations.

Parallelisation over multiple channels The number of available channels depends on the selected instruction set and the bit width. The SSE instruction set uses the 128 bit wide XMM registers of the CPU while the AVX instructions set uses the 256 bit wide YMM registers. Based on this, the number of channels is 8 in the 16 bit SSE version, 16 in the 8 bit SSE and 16 bit AVX versions, and 32 in the 8 bit AVX version.

Score ranges and saturation arithmetic For the 8 and 16 bit searches the available score ranges are -128 to $+127$ using 8 bit and $-32\,768$ to $+32\,767$ using 16 bit. If an alignment score exceeds these, it is recalculated using a higher bit width to get an exact result, due to the greater score range. To prevent wrong results from such over- or underflows, saturation arithmetic is used in combination with an overflow detection. Saturation arithmetic prevents a wrap around, that occurs in modular arithmetic, which is usually implemented in programming languages. Using modular arithmetic with unsigned 8 bit values, adding 200 and 70 results in 14, using saturation arithmetic the result is 255, which corresponds to the upper unsigned limit.

Using Intel's intrinsics, one can choose between saturation arithmetic and modular arithmetic for all basic arithmetic functions. Libssa uses only saturation arithmetic when operating on vectors.

One property of the alignment scores, of the Smith-Waterman algorithm is, they can never be lower than 0. This makes only half of the score range usable, when using signed arithmetic. To use the full range, one has to either use unsigned arithmetic, or treat the lower limit as 0 and convert the score back afterwards.

Using unsigned arithmetic one has to add a bias to the score matrices to convert the values to unsigned numbers. This bias is then subtracted every time after applying a value of the score matrix to maintain the actual values in the score matrix.

Using signed arithmetic, one treats the lower limit as 0 during the computation. Afterwards the final score is converted back. For using the lower limit as 0, the alignment matrix is initialised with the lower limit instead of 0. The rest of the of the computations are unchanged.

Libssa implements the Smith-Waterman using signed arithmetic. This requires one instruction less in each cell computation while the same score matrix can be used for both algorithms, the Smith-Waterman and the Needleman-Wunsch.

Core code for computing one cell The computation on cell level is done on SIMD vectors on up to 32 channels. Listing 4.5 shows the code for computing one cell of the alignment matrix using the Smith-Waterman algorithm with Gotoh's modification (see chapter 2.2.3). At first, the vector of previous values in the `H` matrix is updated with the substitution scores in `V`. Saturation arithmetic ensures that no value exceeds the upper or lower limit of the score range. Afterwards these values are compared to the values in the `E` and `F` matrices to check for the extension of gaps and the highest values of the matrix of each channel are kept. The scores in the current cell are then compared to the maximum scores in `S` of the current alignment. If a value in a channel is higher, `S` is updated to the value of the corresponding channel in `H`. `N` saves the scores of the current cell to be used as `H` values in the computations of the following cells. Afterwards, the values in `E` and `F` are updated for the computations of the following cells.

This implementation somewhat differs from the description of the algorithm. The original algorithm, as described earlier, prepares the scores, checks for the maximum and stores the maximum in the current cell. This implementation initialises the first line and row separately before the alignment. During the computation of each cell, it uses the previously set values and sets the values for the next cells. This way each cell is computed the same way, without having a separate implementation for the first row and column. Compared to the description in chapter 2.2.3, this implementation omits the cases for $i = 0$ and $j = 0$. These are done before, as the initialisation of the first row and column.

Listing 4.5: Core code for computing one cell of the Smith-Waterman algorithm

```
1  #define ALIGNCORE(H, N, E, F, V, QR, R, S )                   \
2   H = _mmxxx_adds_epiYY(H, V);          /* H = H + V       */ \
3   H = _mmxxx_max_epiYY(H, F);           /* H = MAX(H, F)   */ \
4   H = _mmxxx_max_epiYY(H, E);           /* H = MAX(H, E)   */ \
5   S = _mmxxx_max_epiYY(H, S);           /* S = MAX(H, S)   */ \
6   N = H;                                /* N = H           */ \
7   H = _mmxxx_subs_epiYY(H, QR);         /* H = H - QR      */ \
8   F = _mmxxx_subs_epiYY(F, R);          /* F = F - R       */ \
9   F = _mmxxx_max_epiYY(F, H);           /* F = MAX(F, H)   */ \
10  E = _mmxxx_subs_epiYY(E, R);          /* E = E - R       */ \
11  E = _mmxxx_max_epiYY(E, H);           /* E = MAX(E, H)   */
```

Listing 4.6 shows the computation of one cell of the alignment matrix, using the Needleman-Wunsch algorithm with Gotoh's modification. The difference to the implementation of the Smith-Waterman are the overflow detection and the retrieval of the alignment score. The alignment score of the Smith-Waterman algorithm is the maximum score of the whole alignment matrix (line 5 in listing 4.5). The alignment score of the Needleman-Wunsch algorithm is the last computed value of the alignment matrix, which is retrieved outside the cell computation. The Smith-Waterman algorithm does not need an overflow detection at the lower end. The overflow detection at the upper end is done using `S`, the maximum score of the alignment. For the overflow detection in the Needleman-Wunsch, two SIMD vectors were added, `h_min` and `h_max`. These store the minimum and maximum values computed in the alignment of a search block (line 5 and 6 in listing 4.6).

Listing 4.6: Core code for computing one cell of the Needleman-Wunsch algorithm

```
1  #define ALIGNCORE(H, N, E, F, V, QR, R, H_MIN, H_MAX)          \
2   H = _mmxxx_adds_epiYY(H, V);          /* H = H + V       */ \
3   H = _mmxxx_max_epiYY(H, F);           /* H = MAX(H, F)   */ \
4   H = _mmxxx_max_epiYY(H, E);           /* H = MAX(H, E)   */ \
5   H_MIN = _mmxxx_min_epiYY(H_MIN, H);   /* underflow check */ \
6   H_MAX = _mmxxx_max_epiYY(H_MAX, H);   /* overflow check  */ \
7   N = H;                                /* N = H           */ \
8   H = _mmxxx_subs_epiYY(H, QR);         /* H = H - QR      */ \
9   F = _mmxxx_subs_epiYY(F, R);          /* F = F - R       */ \
10  F = _mmxxx_max_epiYY(F, H);           /* F = MAX(F, H)   */ \
11  E = _mmxxx_subs_epiYY(E, R);          /* E = E - R       */ \
12  E = _mmxxx_max_epiYY(E, H);           /* E = MAX(E, H)   */
```

Processing database sequences in blocks The macros for the cell computations are called during the alignment of a search block to the query sequence. For each query residue, the macros are executed 4 times, once for each position in the search block. Chapter 5.2.1 further discusses the size of the search blocks and the two states.

This part is implemented the same way in both algorithms, in the functions `aligncolumns_first` and `aligncolumns_rest`. These functions implement the two states as described in chapter 3.3.3. `Aligncolumns_first` implements the state where new sequences start while `aligncolumns_rest` the other one.

Initialisations for new sequences One difference of both algorithms is the initialisation of the `H`, `E`, and `F` values for new sequences. The Smith-Waterman algorithm initialises these to 0, while the Needleman-Wunsch algorithm initialises these with the costs for a gap.

The computation at this step is done on multiple database sequences, as shown in figure 3.5 on page 35. Here libssa uses masked vector operations to initialise only the new sequences.

Figure 4.1 shows the operations to initialise selected channels in the Smith-Waterman algorithm. Here the mask `M` is set to the lower limit of the bit width in all channels that are reset, and to the upper limit otherwise. The channels in the `H` vector are then reset by computing the minimum between the channels in `M` and `H` . The result is afterwards stored in `H`. The same is done for the `E` values.

H	4	8	56	3	6	67	93	43
M	128	128	-127	128	128	-127	-127	128
min	4	8	-127	3	6	-127	-127	43

Figure 4.1: Initialisation of selected channels in the Smith-Waterman algorithm using the mask `M`. Initialised channels are shown in yellow.

Figure 4.2 shows the operations to initialise selected channels in the Needleman-Wunsch algorithm. Here the mask `M` is set to the unsigned upper limit of the bit width in all channels that are reset and to 0 otherwise. The channels are then reset by subtracting the mask from the `H` vector. The subtraction is done unsigned and in saturation arithmetic. Afterwards, the gap costs are added to all channels that were reset. For this a second mask is created. This mask is set to the gap costs for the current position in the query sequence in all channels that are initialised. The same is done for the `E` vector.

In both algorithms the masks are initialised when the channels are refilled. At this point all channels are processed sequentially. Hence the algorithm knows which channels contain new sequences. During the

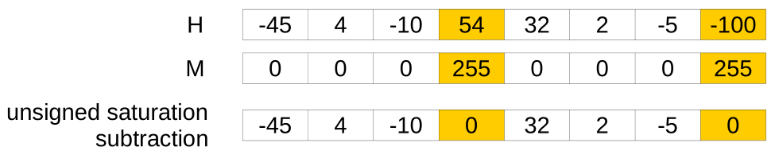

H -45 4 -10 54 32 2 -5 -100
M 0 0 0 255 0 0 0 255
unsigned saturation subtraction -45 4 -10 0 32 2 -5 0

Figure 4.2: Initialisation of selected channels in the Needleman-Wunsch algorithm using the mask `M`. Reset channels are shown in yellow.

actual alignment, the algorithms compute on all channels uniformly in parallel.

Storage of the `H`, `E`, and `F` values Another aspect of efficiency, besides throughput, is memory usage. Here one wants to keep as little data as needed. This is important for sequences alignments, since the alignment matrices grow exponentially with the lengths of the aligned sequences. Here libssa reduces the memory usage by storing only those cells, which are needed to compute the next cells.

Each value in the matrices `H`, `E`, and `F` depends on a previous value. `H(i, j)` depends on `H(i-1, j-1)`, `E(i, j)` depends on `E(i, j-1)`, and `F(i, j)` depends on `F(i-1, j)`, with `i` being the index in the query sequence and `j` being the index in the database sequence. To reduce the memory usage, only those values are kept which are needed to compute the next cells. Since the computations are done column-wise on 4 database residues in a row, the kept values are the last computed column of `H` and `E` to continue the alignments from the last search block. Additionally 4 cells of `H` and `F` are kept for the values from above, which are initialised as values of the first row.

Temporary score profiles The vector `V`, shown in listing 4.5 and 4.6, stores the substitution scores of query and database residues. These are pre-computed and stored in a temporary score profile. The advantages here are the possible re-use of each score vector and the optimised computation of it.

Nucleotide sequences consist of 16 different symbols while protein sequences have 28 different symbols. As a consequence there are only 16 and 28 different substitution scores exist, respectively, for aligning one database residue to all query residues. Hence pre computing these vectors of substitution scores allows for reusing them during the alignment of the whole query sequence to the database residues of one search block.

The temporary score profile provides the data for the `V` vectors. This profile contains 32 vectors for each of the 4 positions in the search block. The 32 vectors match the 32 possible symbols in the score matrices which resemble the symbols occurring in the sequences. 32 was chosen as it is the smallest two's complement greater or equal to the number of symbols in nucleotide and amino acid sequences. Padding the matrices to the next two's complement supports the creation of the score profile on SIMD vectors as one can see in the next paragraph. Loading the substitution scores for one query residue is now one load operation of the matching vector of the score profile.

Figure 3.6 on page 36 illustrates a way of filling the score profile. The actual implementation loads lines instead of columns and stores them transposed. Loading the lines, transposing, and storing them is done in SIMD vectors. This way all channels are processed at the same time which would not be possible if the values would be copied column-wise. The matrices are stored line-wise in memory, hence loading a column requires multiple load operations in contrast to one load operation for loading a line. Figure 4.3 shows the different ways of filling the score profile for the implemented versions. In the simplest case, the 8 bit search using AVX instructions, 32 lines are loaded and transposed in one pass. In the other implementations, the transpose operation is split depending on the number of integers fitting into one register. Selecting the matching lines and storing them in transposed representation is done 4 times, once for each position in the search block.

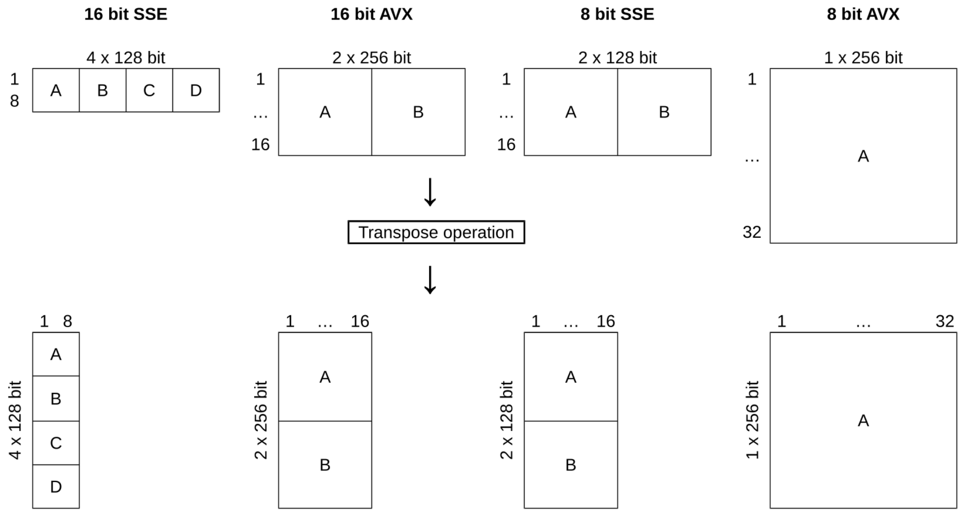


Figure 4.3: Illustration of the transpose operations performed for different bit widths and SIMD instruction sets. Each square, labelled with A to D, is a transpose operation.

Here one can see the importance of the matrix dimensions of 32 by 32 values. The transpose operations shown in figure 4.3 would be the same when the dimensions would be 28 by 28, the minimum size for amino acid sequences. The difference is that each last transpose operation would then be padded to 32. Padding the matrices from the beginning reduces the work done when the score profiles are build.

The implementations for creating the score profile are found in the `dprofile_fill` functions in the `search_util` files. These implement the transpose operations using a series of unpack operations as illustrated in figure 4.4. Here, the integers of each vector are shuffled between the vectors to reorder them in a column and thus transposing the lines. There are two kinds of unpack operations, one on the lower half and one on the upper half of the vectors. Both store the data interleaved in a new vector.

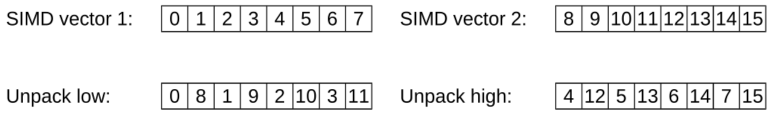


Figure 4.4: Visualisation of the unpack operations on the low and high halves of two SIMD vectors. Unpacking is done by taking the low or high halves of two vectors and storing them interleaved in a new vector.

In the implementations, one can notice the use of multiple `for` loops. These are all declared with a constant number of iterations, so the compiler can optimise the code by unrolling the loops. The source code on the other hand becomes shorter and better readable using the loops.

Loading sequences and storing alignment scores The temporary score profile provides the substitution scores for the database residues in one search block. The step before the computation of it prepares this search block by loading the database sequences into the channels and setting the search block to the current position.

Figure 3.5 on page 35 shows the iterations of a search block over the sequences in multiple channels. The block is moved by loading the next 4 residues of each sequence into it. The handling of entire sequences follows the flow chart shown in figure 3.4 on page 35. The action shown in green loads the residues into the blocks, while the yellow actions prepare the sequences in the channels.

The preparation of sequences in the channels covers loading new sequences, storing the alignment scores of finished sequences, and the overflow handling. The handling of overflows is described in the next part, while this paragraph focuses on the loading and storing.

The database sequences are loaded sequentially. They are stored in contiguous memory regions, unrelated to each other. Hence it is not possible to load one residue of each database sequence into one vector in one step. SSE and AVX implement load operations for contiguous memory only. Although AVX2 implements gather operations to load non contiguous data, these are restricted to 32 and 64 bit integers and single and double precision floating point numbers. Therefore the sequences are loaded sequentially for both instructions sets.

When the alignment of a sequence in a channel is finished, the score is stored in the thread-local min-max-heap as described in chapter 4.3.2. This is done in the loop iterating over the channels to move the search block and exchange sequences. The finished sequence is then removed from the channel and exchanged with a new sequence until all sequences of the current chunk are processed.

Overflow detection Another reason, for loading a new sequence besides a completed alignment is an overflow of the alignment score. This occurs when the score computed for a cell of the alignment matrix exceeds the score range.

In sequence alignments, overflows can occur at any cell of the matrices. They can easily be hidden by following calculations, such that the final score does not exceed the limit even if one score in between exceeds it. The final score is then, of course, incorrect. The detected overflows are per search block. Here the highest and the lowest computed score are collected and afterwards checked for overflows.

In saturation arithmetic overflows do not wrap around. Calculations exceeding the upper or lower limit result in a value equal to the corresponding limit. Overflows are now detected by comparing the collected highest and lowest scores to the upper and lower limit of the bit width. Scores equal to the respective limit are treated as overflows. Each sequence where an overflow occurred is added to an overflow list to re-align it with the next higher bit width.

Collecting the minimal and maximal scores is realised differently for the Smith-Waterman and the Needleman-Wunsch algorithm. In the Smith-Waterman algorithm overflows are only possible at the upper limit since the scores can never be lower than 0. The overflows at the upper limit are detected using the score of the local alignment, which is the maximal score of the whole alignment matrix. Listing 4.7 shows the instructions used to detect overflows and to force a sequence change, in the affected channel. `Score_max` holds the maximum value of the current score range, while `S.v` holds the maximum score of the alignment matrix. The intrinsic `_mmxxx_cmpeq_epiYY` sets each overflown channel in `overflow.v` to $0xff$. Afterwards `_mmxxx_movemask_epi8` adds up the most significant bits of `overflow.v`, resulting in a value greater than 0 if at least one channel is overflown.

Listing 4.7: Instructions for detecting overflows in the Smith-Waterman algorithm.

```
overflow.v = _mmxxx_cmpeq_epiYY( S.v, score_max );
change_sequences |= _mmxxx_movemask_epi8( overflow.v );
```

In the Needleman-Wunsch algorithm, overflows are detected at both limits. Listing 4.8 shows the instructions used to detect these, and to force a sequence change in a channel. `Score_min` and `score_max` are vectors holding the minimal and maximal possible values of the current score range, while `h_min` and `h_max` hold the last computed minimum and maximum scores. The instructions `_mmxxx_cmpgt_epiYY` and `_mmxxx_cmpeq_epiYY` compare the last computed minimum and maximum scores to the respective limit. Each overflown channel in `overflow.v` is set to $0xff$ and afterwards `_mmxxx_movemask_epi8` adds up the most significant bits of `overflow.v`, resulting in a value greater than 0 if at least one channel is overflown.

Listing 4.8: Instructions for detecting over- an underflows in the Needleman-Wunsch algorithm.

```
overflow.v = _mmxxx_cmpgt_epiYY( score_min, h_min );
overflow.v = _mmxxx_or_si( _mmxxx_cmpeq_epiYY( h_max, score_max
    ), overflow.v );
change_sequences |= _mmxxx_movemask_epi8( overflow.v );
```

The variable `change_sequences` states, whether a sequence in at least one channel needs to be changed. This flag is evaluated in each iteration of the main loop processing the sequences of a chunk. It is used to switch between the two states described in chapter 3.3.3.

4.5 Computing alignments

The implementation, for computing alignments, was partly taken from the SWIPE and the SSW projects. Computing the directions is implemented as in SWIPE while computing the CIGAR string is implemented as in SSW.

Alignments are computed on multiple threads, where each thread does the computation in 64 bit mode without using SIMD instructions. The alignments are performed based on the result of the database search, on the highest scoring pairs of query and database sequences. To process these, each thread receives a pair at a time, computes the CIGAR string, and stores it in a result list. This is done until all pairs are processed. Afterwards the local results are combined in the `manager` module.

For local alignments, the region of the alignment is found first. This region is defined as the part of the alignment matrix between the highest scoring cell and a trace back to the first found cell with a value of 0. If the highest score occurs multiple times in the alignment matrix, the first one is used. For global alignments this is not done, since the region of interest is the whole alignment.

The CIGAR string is computed during a trace back from the end of the alignment regio, to the start. For this operation a direction matrix is computed first as already described in chapter 2.2.3. Afterwards the CIGAR string is computed while tracing back the directions. The CIGAR string is then stored in the compressed form and is computed using only match (M), delete (D), and insert (I) actions.

The computation of alignments is not optimised. It is implemented in 64 bit mode and stores the whole direction matrix in memory. For two sequences of length m and n, this matrix uses $m * n$ bytes of memory. By replacing the file `align.c` and `cigar.c`, one could replace this implementation with, for example, a linear memory aligner as described by Myers and Miller [1988]

4.6 Testing

During the development of the library, a test suite was implemented. It contains unit tests for all non static functions, a set of integration tests for the external database library, and a set of system tests. Additionally a set of programs was developed for measuring the performance. These are described in chapter 4.7 and are used to execute performance tests.

The unit tests are developed to cover as many statements and branches as possible. They include positive and negative test cases. The integration test cases cover the API, which is used to integrate external databases (`libssa_extern_db.h`), while the system tests cover the functions, implemented by the public API (`libssa.h`). They test the interaction of a user with the API.

Another aspect covered by unit tests is the functionality of sequence alignments. Here one has to make sure that the computation of sequence alignment is correct. This includes the alignment scores as well as the CIGAR strings. The base of these is a set of local and global sequence alignments where the results are confirmed by hand. These tests include nucleotide and protein sequences of up to about 15 residues. The results for longer sequences are confirmed by computing these on the different implementations of the algorithms in libssa. The 8 and 16 bit implementations are fundamentally different from the 64 bit implementations. An error in one implementation does not need to be present in another one. Chapter 5.1.7 discusses more thoroughly the challenge of finding meaningful tests.

All test cases are implemented using the Check unit testing framework, version 0.9.10[8], available under the LGPL v3 license [9]. For each header file in the source folder a test file in the test folder exists, which implements the test case for the corresponding header file. These are named like the header file with `test_` as prefix while the folder structure in the test folder resembles the folder structure in the source folder. Additional test data, like database and query files, is located in the `testdata` sub folder of the test folder.

[8]http://check.sourceforge.net/, last visited 1.4.2015

[9]http://www.gnu.org/licenses/lgpl.html, last visited 1.4.2015

Each test file implements a function for registering the test case in the test suite. Their declarations are collected in the header file `tests.h` and called in the function `libssa_suite` of the file `check_libssa.c`. This function collects all test cases to create the test suite, which is then executed.

Each test case contains a number of positive and negative tests, checking the correctness of the public functions in the corresponding header file. For each test, a new child process is forked off. This allows a signal or early exit to be caught and reported without taking down the whole test suite execution. Additionally a time limit is set, for the whole test case to prevent infinite runtimes. All test cases run with the default time of 4 seconds except for the system test `test_bigger_databases.c`. Here the time-out is set to 30 seconds. Depending on the system the tests are run on, this time-out needs to be set to a higher value.

4.7 Measuring performance

A set of tools is implemented, to measure the performance of the library. The focus here is on the runtime performance. They are located in the `benchmark` sub folder of the library.

This chapter describes some implementation details of the implemented tools. The execution of these is described in chapter B, which gives a detailed introduction to reproducing the performance results described in chapter 5.1.

All of the implemented tools measure the wall clock time. Using CPU clock ticks as measurement would make it easier to compare different runs of the library with the same configuration since CPU clock ticks are independent from other programs running on the same computer. On the hand this measure would make it more difficult to compare different configurations. For example, if a run with multiple threads is compared to a run with one thread, the number of CPU clock ticks is higher at the run with multiple threads, although the wall clock time is lower.

Overall runtime The overall runtime is measured using the example application for libssa and a set of shell scripts. The example application runs a database search configured by command line parameters, while the shell scripts call the application in different configurations, measuring and logging the runtime.

The important shell scripts are `runner.sh` and `aligner_comparison.sh`. The first one executes a command a given number of times and prints the runtime to both `stdout` and a file. The second shell script runs the example application in a couple of selected configurations. Each run is carried out a given number of times using `runner.sh`. The results are printed to a file as comma separated values. The script `aligner_comparison.sh` runs a benchmark for different alignment tools to compare their runtime. The additional shell scripts are wrappers for executing the various tools.

The runtime is measured using the Linux command `date` with a precision of nanoseconds[10]. The measured time is the difference between a call to `date` before starting the program and a call to `date` after the program is finished.

Runtime of the alignment functions only The runtime of only the database searches is measured using different tools. They are implemented in C and call libssa directly. Each tool iterates through a set of configurations and runs a database search in each of it.

The measured runtime is limited to the runtime of the database searches only, excluding the computation of the CIGAR strings. Also the time spent setting up the database searches is excluded. The runtime is measured using the function `gettimeofday` from the header file `sys/time.h`[11]. This function returns the wall clock time with a precision of microseconds. The measured time is the difference between a call to `gettimeofday` before starting the database search and a call to `gettimeofday` after the program is finished.

[10]http://man7.org/linux/man-pages/man1/date.1.html, last visited 5.3.2015
[11]http://man7.org/linux/man-pages/man2/gettimeofday.2.html, last visited 5.3.2015

Chapter 5

Results and discussion

This chapter evaluates the implementation of libssa and discusses various improvements that were made during the implementation. Furthermore it evaluates the performance improvements of using the AVX instruction set over the SSE instruction set. The first part, chapter 5.1, evaluates the performance of libssa, while the second part, chapter 5.2, discusses the optimisations that were applied to libssa. At the end chapter 5.3 gives an overview of the source code, which was taken from different open source projects and re-used in libssa.

5.1 Evaluating performance

This chapter introduces and evaluates performance results collected using the set of benchmarks described in chapter 4.7. The results were collected as CSV (comma separated value) files and evaluated using the programming language R. Appendix B describes how to reproduce them.

All results were gathered on a Dell laptop equipped with an Intel(R) Core(TM) i7-4800MQ CPU, 8 GB of memory, and a 500 GB hard disk. The CPU has 4 physical cores and 8 hyper-threads. The processor frequency is 2.7 GHz with a maximum turbo boost frequency of 3.7 GHz[1]. The operating system is a 64 bit Linux Mint version 17 running the standard Linux kernel version 3.13.0.

5.1.1 Collecting results

Database searches performed with libssa can be configured in several ways. Each parameter can be can used in combination with any other parameter. As an exception, setting the SIMD capabilities has no effect on the 64 bit searches and the 8 bit searches requires at least SSE4.1. Following is a list of all parameters and their possible values:

- Number of used threads: $n \in \mathbb{N}$

[1]http://ark.intel.com/products/75128/Intel-Core-i7-4800MQ-Processor-6M-Cache-up-to-3_70-GHz, last visited 3.4.2015

- Chunk size: $n \in \mathbb{N}$
- SIMD capability: SSE2, SSE4.1, AVX2
- Bit width: 8, 16, 64
- Search algorithm: Needleman-Wunsch (NW), Smith-Waterman (SW)
- Database: any DNA/RNA or protein database
- Query sequence: any FASTA formatted DNA/RNA or protein sequence
- Gap costs: -128 to $+127$ for opening and extending
- Substitution matrix: BLOSUM, PAM, user defined

Libssa sets no limits to the size of the database. Since it is processed in chunks, it can have an arbitrary size. Chapter 5.1.5 evaluates this parameter and gives advice on optimal values for it. The situation is similar for the number of threads. It is only limited by the operating system and the hardware, however, chapter 5.1.4 gives advice on an optimal number of threads.

While measuring the performance, only a difference between SSE and AVX is made. Benchmarks are performed for SSE searches are done with a SIMD capability of SSE4.1. The intention here is to show the improvement of using the wider AVX vectors, not to compare the different versions of the SSE and AVX instruction sets.

Two different databases are used in the tests. The UniProtKB/Swiss-Prot database[2] in version 3/2015 and the Rfam database[3] in version 11. The Swiss-Prot database is a subset of manually annotated protein sequences of the UniProtKB database. It contains 547 964 sequences with a total number of 195 174 196 amino acids. The Rfam database contains RNA sequences of various families. It contains 383 004 sequences with a total number of 52 830 109 nucleotides.

Most of the tests are performed on protein sequences only, using the UniProtKB/Swiss-Prot database. This is done, since internally both kinds of sequences are treated the same. In the current implementation, the only difference between nucleotide and amino acid sequences is the size of the alphabet. The nucleotide alphabet contains 16 symbols while the amino acid alphabet contains 28 symbols (see chapter 4.1.2). The algorithms themselves do not distinguish between both kinds of sequences and the size of the score matrices is the same for both as well. The implemented test suite ensures the correct handling of nucleotide sequences, while the benchmark in chapter 5.1.5 shows some performance data for these.

[2]http://www.uniprot.org/uniprot/, last visited 3.4.2015

[3]http://rfam.xfam.org/, last visited 3.4.2015

5.1.2 Base test run

To asses the performance of libssa, a number of test runs are carried out. In the beginning, a basic test run is executed to get an overview of the performance which is later used to reduce the number of tested configurations in the following benchmarks.

Configuration This test run only measures the runtime of the database searches, excluding the initialisations steps and the computation of the CIGAR strings. The following list shows the the configuration dates that are used in all combinations:

- Number of threads: 1, 8
- Chunk size: 1000 sequences
- SIMD capabilities: SSE, AVX
- Bit width: 8, 16, 64
- Search algorithm: Needleman-Wunsch, Smith-Waterman
- Database: UniProtKB/Swiss-Prot
- Query sequences: *O74807*, *P19930*, *Q3ZAI3*, *P18080*
- Gap costs: -3 opening and -1 extending a gap
- Substitution matrix: BLOSUM50

The gap gap opening and extension costs of -3 and -1 used in this test run are an example of possible gap costs. They have only little influence the performance, which is discussed in chapter 5.1.3.

The database searches are run 10 times in each of these configurations. This is done to detect variations induced by the system the test is running on. 4 query sequences of varying length are tested to get a better picture of the performance for different query sequence lengths. The lengths of the used sequences range from 110 to 513 residues.

Variation of timing results Executing a program on a computer multiple times often results in variations of the runtime. These variations can be a sign of inefficient programming, or they can be induced by other applications, running on the same system, the operating system itself, or the hardware. The influence of other running programs can be reduced by simply terminating them. The effect of other influencing factors is evaluated using box plots, showing the variation in the runtime of each of the 10 times each configuration is run.

Figure 5.1 and 5.2 show the box plots illustrating the variations over the runtimes. The data is here reduced to the query sequence *P19930*. Other sequences exhibit a similar variation. The highest standard deviation for the searches on 1 thread is 1.63 seconds using the 64 bit Needleman-Wunsch. For the searches on 8 threads it is 0.08 seconds using the 64 bit Smith-Waterman.

All box plots exhibit only little to no variation, which indicates that the runtime is less influenced by the system the database searches are executed on. Due to this, the following plots only show the average runtime over 10 runs.

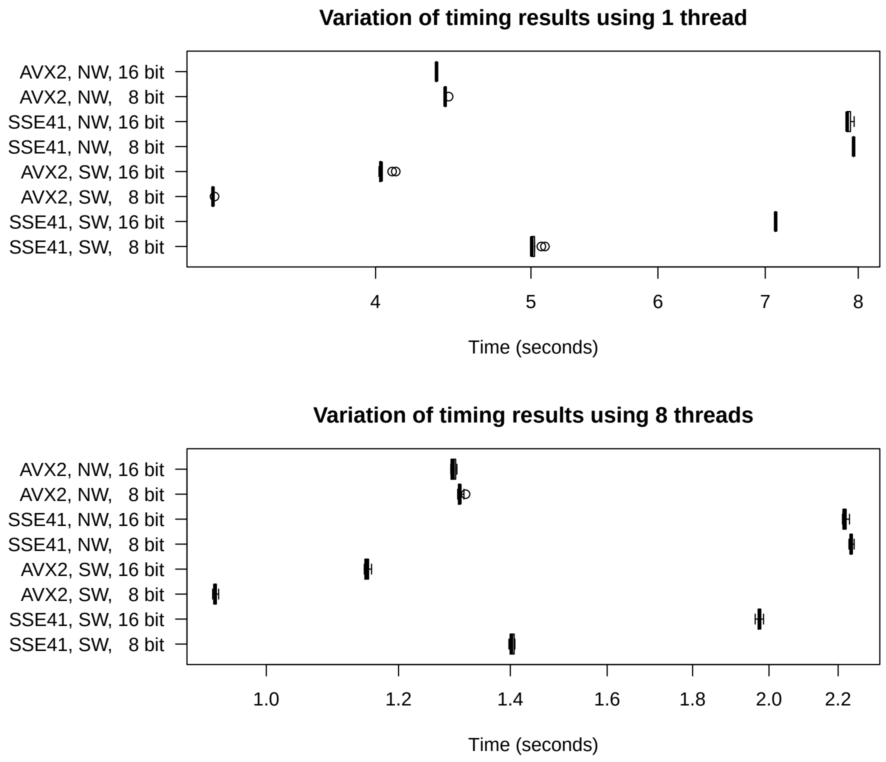


Figure 5.1: Boxplots of the variations in the runtime, for the 8 and 16 bit searches, in different configurations. The circles show outliers to the runtimes while the vertical bars are collapsed boxplots illustrating the little variation.

Results for 8/16 bit and AVX/SSE Figure 5.3 shows the runtimes for the 8 and 16 bit searches computed on SSE and AVX. These exhibit a varying correlation between the 8 and 16 bit searches for shorter and longer query sequences. 8 bit Needleman-Wunsch searches seem to be slower for shorter sequences than the 16 bit counterparts, while they are equally fast for longer sequences. The Smith-Waterman searches show an opposite behaviour, being faster for shorter queries and slower for longer queries.

The base test run covers only 4 query sequences with lengths ranging from 110 to 513 residues. The next section discusses a second test run which

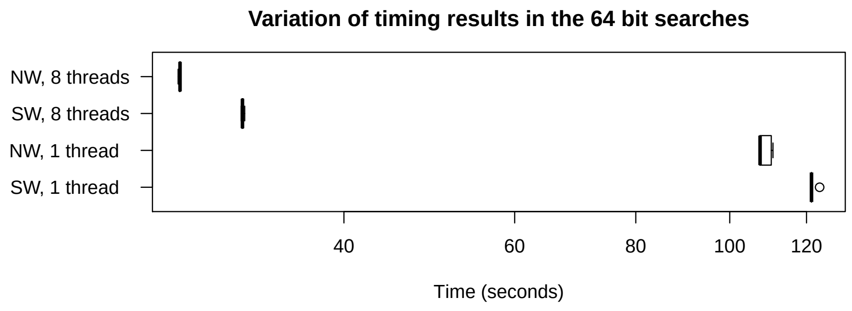


Figure 5.2: Boxplot of the variations in the runtime, for the 64 bit searches, in different configurations. The circles show outliers to the runtimes while the vertical bars are collapsed boxplots illustrating the little variation.

covers a wider range of query lengths. This test run evaluates the varying behaviour of the results shown here.

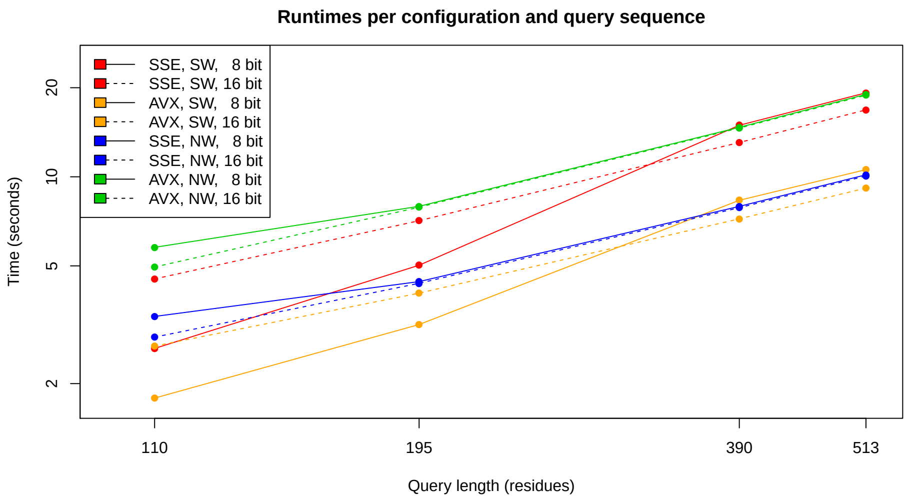


Figure 5.3: Chart of the runtime of different database searches per query sequence. Each curve shows a run with a different configuration.

5.1.3 Query lengths, bit widths, and SIMD capabilities

The results of the base test run suggest an influence of the length of query sequences on the performance of other parameters like the bit width and the used SIMD capabilities. The expectation here is that the 8 bit searches are faster than the 16 bit searches, with the same statement holding for the computations on the wider AVX registers with respect to the computation on the SSE registers. The length of the query sequence itself should effect the performance, since the size of the computed alignment matrices is based on it and with this the number of computed cells per alignment.

This benchmark evaluates the performance of different parameters in correlation to different query lengths. The tested parameters are the bit width, both algorithms, and the SIMD capabilities.

Configuration This test run uses 36 query sequences of lengths ranging from 24 to 5 478 amino acid residues. These are taken from the UniProtKB/Swiss-Prot database. The benchmark is then run on 8 and 16 bit on 8 threads. The rest of the configurations are the same as in the base test run (see chapter 5.1.2).

Results Figure 5.4 shows the results of the Smith-Waterman database searches for 8 and 16 bit and both instruction sets SSE and AVX. The 16 bit searches show a constant gain in runtime over growing query lengths. The 8 bit searches, on the other hand, start with a flat performance curve for short query sequences, up to around 200 residues. At around 200 residues, the performance drops below the 16 bit performance. From there on the 8 bit performance curve follows the 16 bit curve with a slightly higher runtime.

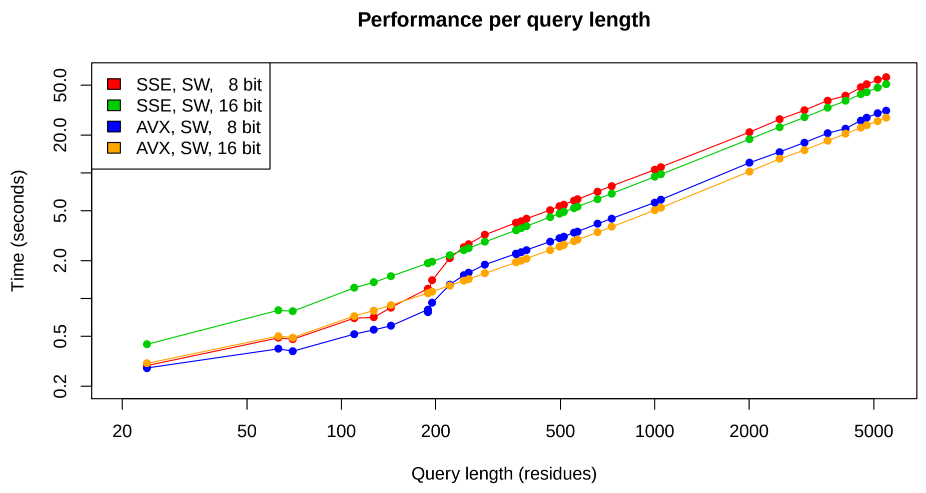


Figure 5.4: Visualization of the runtime of Smith-Waterman database searches using query sequences of varying length.

Figure 5.5 shows the results of the Needleman-Wunsch database searches, for 8 and 16 bit and both instruction sets SSE and AVX. Here the 16 bit performance curve starts rather flat until around 100 residues and grows from there on linear to the query length. The 8 bit searches are slower than the 16 bit searches until a query length of around 120 residues. From there on both bit widths are equally fast.

Tables 5.1 and 5.2 show the number of overflows per query length for the Needleman-Wunsch and the Smith-Waterman algorithm. Both tables are

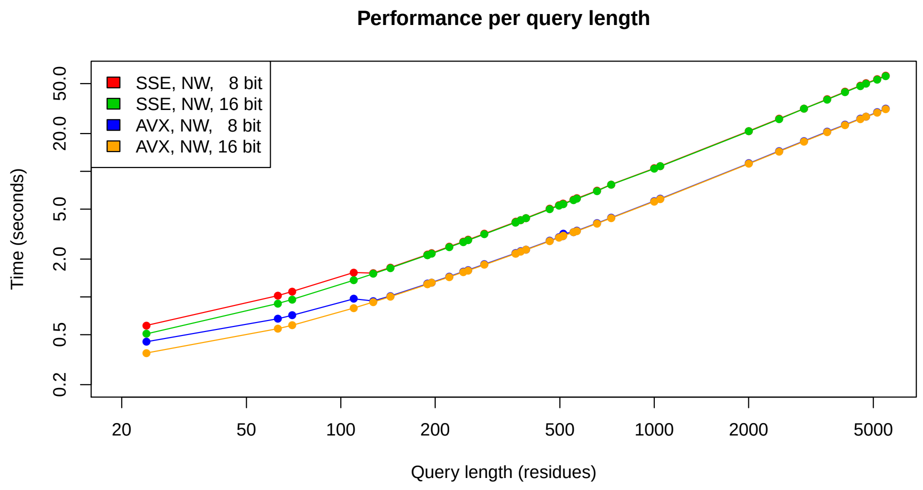


Figure 5.5: Visualization of the runtime of Needleman-Wunsch database searches using query sequences of varying length.

reduced to the query lengths where the performance curves, of the respective database searches, change their behaviour.

Query length	24	70	110	127	144
Overflows	460 448	460 456	461 645	542 349	547 964

Table 5.1: Number of overflows per query length from 8 to 16 bit, using the Needleman-Wunsch algorithm.

Query length	189	195	222	246	255
Overflows	5 516	55 916	163 402	239 087	257 297

Table 5.2: Number of overflows per query length from 8 to 16 bit, using the Smith-Waterman algorithm.

Figure 5.6 and 5.7 depict the improvement ratio of database searches on AVX registers and 8 bit searches, respectively. The ratios are computed by dividing the runtimes of the AVX and the 8 bit searches by the runtimes of the SSE and 16 bit searches. A ratio of 1 indicates that both configurations are equally fast, while a lower ratio shows that the configuration, which is supposed to be faster, is actually slower. A higher ratio indicates an improvement in the performance for the more advanced configuration.

In figure 5.6, all curves exhibit a rather constant ratio of about 1.8 for query sequences longer than 250 residues. For shorter query sequences the curves for the 8 and 16 bit Needleman-Wunsch and the 16 bit Smith-Waterman show an increase in the ratio from about 1.4 to 1.8. The curve for the 8 bit Smith-Waterman searches starts lower at about 1.04 and grows from there on to about 1.8, at a length of around 250 residues.

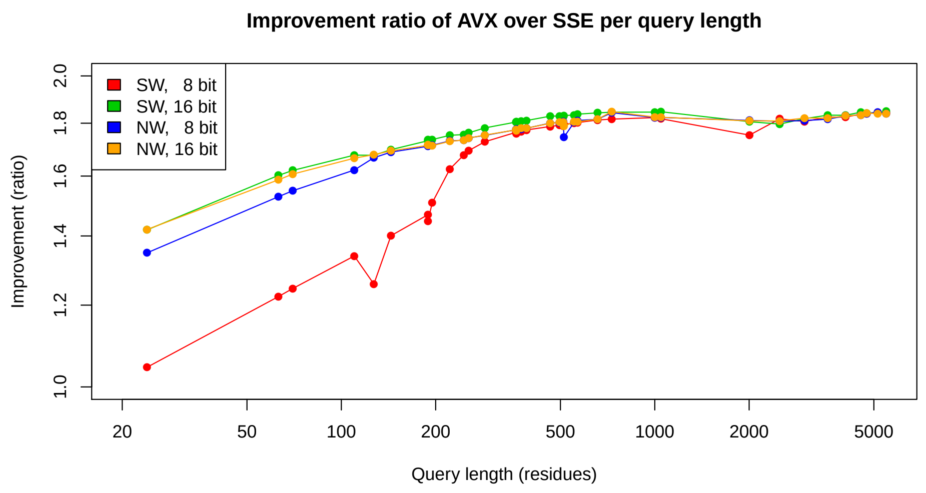


Figure 5.6: Visualization of the improvement ratio of database searches using AVX over the SSE instruction set for query sequences of varying length.

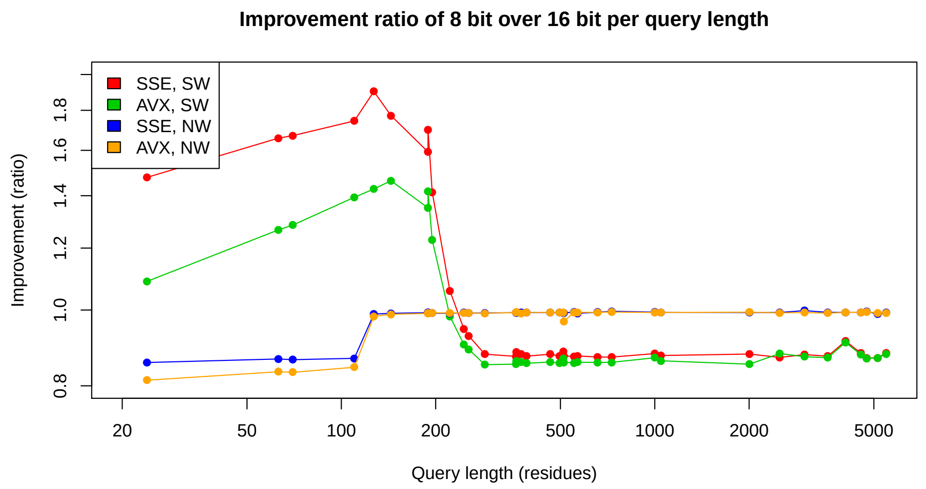


Figure 5.7: Visualization of the improvement ratio of 8 bit database searches for query sequences of varying length.

The curves in figure 5.7 show a different behaviour for the Smith-Waterman and the Needleman-Wunsch database searches. For shorter query lengths, until about 200 residues, the 8 bit Smith-Waterman searches are performing better, while for longer query sequences they perform worse, than the 16 bit searches. The 8 bit Needleman-Wunsch searches perform worse for shorter query lengths until about 100 residues, while they perform about as good as as the 16 bit searches for longer query sequences.

Discussion Both diagrams, in figure 5.4 and 5.5, show a correlation of the runtime performance to the query length regarding the bit width and the algorithm. The performance improvement of AVX over SSE is about constant for longer query sequence lengths while it varies for shorter queries.

In figure 5.6 one can see that the performance of the database searches generally improves when the computation is done on the wider AVX registers. The ratio of improvement starts low, increases until a length of around 250 residues, and is constant from there on.

A reason for the lower ratio of improvement is the computation of the score profile. The runtime for computing one score profile is constant over the query lengths. This means for shorter query lengths the computation takes up a bigger portion of the overall runtime. In the computation on AVX this portion is even bigger since the throughput of the database search is twice as high. The tables 5.3 and 5.4 present the percentages of the four most used functions of 8 bit database searches using the 24 residue long query sequence *P56980*. Here one can see that the `dprofile_fill` functions takes up more CPU cycles in the AVX search than in the SSE search.

CPU cycles	Function
45,77%	`dprofile_fill_8_avx2`
33,38%	`search_8_avx2_sw`
10,56%	`db_read`
8,42%	`us_map_sequence`

Table 5.3: Percentages of the four most used functions, of an 8 bit AVX Smith-Waterman database search, for the query sequence *P56980*.

CPU cycles	Function
46,44%	`search_8_sse41_sw`
34,66%	`dprofile_fill_8_sse41`
9,52%	`db_read`
8,08%	`us_map_sequence`

Table 5.4: Percentages of the four most used functions, of an 8 bit SSE Smith-Waterman database search, for the query sequence *P56980*.

The lower ratio for shorter queries for the 8 bit Smith-Waterman searches is based on the number of overflows. Table 5.1 presents 460 448 overflows for the 24 residue long sequence *P56980* using the Needleman-Wunsch algorithm. Using the Smith-Waterman algorithm, no overflows occur for this sequence. This means for this sequence the 16 bit search takes up the bigger portion of the runtime, when the Needleman-Wunsch is used, which is why the 8 bit Needleman-Wunsch searches show a behaviour closer to the

16 bit searches. The Smith-Waterman searches show a different behaviour, between 8 and 16 bit searches, due to less overflows.

The sequences *P01111* and *P05013* are both 189 residues long. Figure 5.6 and 5.7 both show different ratios for for both sequences, despite the fact that they are of the same length. This is again caused by overflows, with them occurring 21 756 times for the sequence *P01111*, but only 5 516 time *P05013*, resulting in a better ratio for *P05013*.

The improvement of 8 bit searches over 16 bit searches clearly depends on the query length as one can see in figure 5.7. For short query sequence lengths 8 bit Smith-Waterman database searches are faster than their 16 bit counterparts. Needleman-Wunsch database searches show, on the other hand, a diametrical behaviour, being slower for shorter query sequences. The tables 5.1 and 5.2 indicate a correlation to the overflows in the computations from 8 to 16 bit. If most of the alignments are re-computed using 16 bit, it becomes more efficient to do the computation directly in 16 bit.

For the Needleman-Wunsch the minimum length of a sequence at which it is more efficient to do the computation directly in 16 bit can be calculated. This algorithm initialises the first row and column of the alignment matrix with the scores for aligning a sequence to an empty sequence (see chapter 2.2.3). So is, for example, the score for the alignment of the query sequence *O74807*, with a length of 110 residues, to an empty sequence $-3 + 110 * (-1) = -113$, if the gap opening cost -3 is and the gap extension cost -1. This equation, $gapO + length * gapE = score$, for the alignment score of a sequence to an empty sequence can be rearranged to calculate the minimum length, at which a 16 bit search becomes more efficient: $length = (score - gapO) / gapE$. The 8 bit searches underflow at a score of -128. Therefore the minimum length, with a gap opening cost of -3 and a gap extension cost of -1, is $(-128 - (-3)) / (-1) = 125$. The chart in figure 5.7 supports this as the ratio jumps at a length of about 120 residues from around 0.83 to about 1.

The rapid overflow behaviour at 8 bit of the Needleman-Wunsch algorithm is the reason of the high number of overflows for even short query sequences. These overflows occur, among others, when the database sequences are longer than 125 residues, considering the configured gap opening and extension costs of this test run. In the UniProtKB/Swiss-Prot database most of the sequences are longer than 125 residues, as the average sequence length is 356 residues.

In the vectorised database searches the alignments are done column-wise along the query sequence. When the query sequence is too long and it overflows, like described earlier, the overflow happens on the first aligned column. After that column the overflow is noticed, the database sequences in all channels are marked for realignment, and the next set of database sequences is tried. Computing the first column for all database sequences does not take so long compared to the complete alignments. Therefore the ratio between 8 and 16 bit for longer query sequences is almost 1.

Concluding one can say that database searches using an 8 bit score range are only beneficial if the average length of the query and database sequences is short. For the Needleman-Wunsch algorithm the limit is at a length of about 100 residues, while for the Smith-Waterman it is at a length of about 200 residues, using a database with an average sequence length of about 356 residues, such as the UniProtKB/Swiss-Prot database. Additionally one can see that the computation on the wider AVX registers improves the performance of the library. Even with a smaller improvement at shorter query sequence lengths. The lower benefit in those cases could potentially be improved by a more advanced algorithm for computing the score profiles.

5.1.4 Thread counts

Another parameter of libssa is the number of threads the computation is performed on. The expectation here is that the performance increases when the database searches are done on more than one thread. This is evaluated using a benchmark testing a number of constant configurations against different thread configurations. This benchmark is run twice to evaluate the effects of hyper-threading. One run is done with enabled hyper-threading and one with disabled hyper-threading.

Configurations The tested configurations are reduced to the Smith-Waterman database search of the sequence *P18080* in the UniProtKB/Swiss-Prot database. The search is done using the sequential 64 bit implementation and the vectorised 8 bit AVX implementation. Each of these configurations is run with 8 different thread configurations in the test with enabled hyper-threading and 4 different thread configurations in the test with disabled hyper-threading. This setup was chosen since the CPU of the test system implements 4 cores with 8 hyper-threads.

Results Figures 5.8 and 5.9 show the results of the test runs, with enabled and with disabled hyper-threading. In both figures, the variation in the runtime is shown as a box plot, while the blue and orange curves mark the average runtime, per thread configuration, for the 8 and 64 bit searches. The y-axis is shown in log-scale.

Table 5.5 shows the ratios of improvement per test run, for running a database search with the maximum number of threads over the minimum number of threads. These are calculated by dividing the mean runtime with the lower thread count by the mean runtime with the higher thread count.

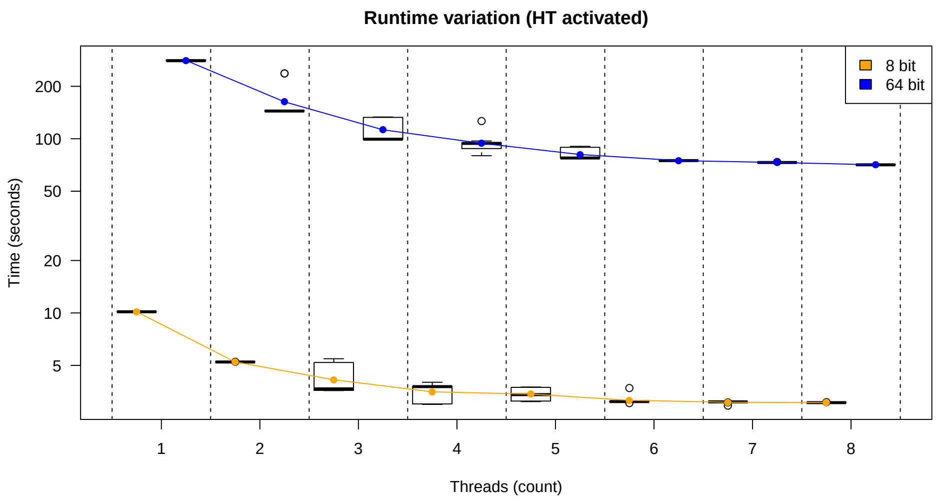


Figure 5.8: The box plot shows the runtime variation, using different thread configurations with enabled hyper-threading. The blue and orange lines mark the average runtime of the 8 and 64 bit search.

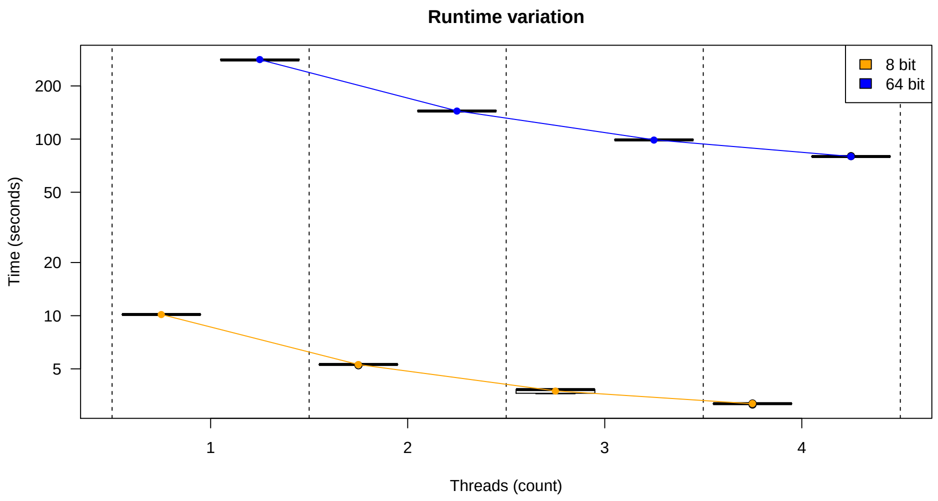


Figure 5.9: The box plot shows the runtime variation, using different thread configurations with disabled hyper-threading. The blue and orange lines mark the average runtime of the 8 and 64 bit search.

	8 bit	64 bit
no HT: 1 vs. 4 threads	3.20	3.53
HT: 1 vs. 4 threads	2.88	2.99
HT: 1 vs. 8 threads	3.32	3.97

Table 5.5: Ratio of improvement per thread count and bit width

Discussion In both graphs one can see that libssa scales good to multiple threads. Although hyper-threading induces a variation to the runtime.

The variation in the test run with enabled hyper-threading is caused by a sub-optimal distribution of some of the threads to the cores and hyper-threads. If two threads are running on the same core, the second thread can only use the execution units the first thread is not using. This results in time spend waiting for the second thread which increases the runtime. The variation occurs since the threads are not always distributed in such a way. This is supported by the variation being present in only the thread configurations, that can be distributed sub-optimally.

The implementation leaves the distribution of the threads to the scheduler of the operating system. Hence the variation shown in these test cases might not occur on every system.

A benchmark of the libssa example program using the benchmarking tool perf[4] shows that the database searches account for about 99% of the runtime, when the searches are done in 64 bit, and for about 87.5%, for 8 bit searches. Using Amdahl's law, one can calculate the theoretical performance gain of using multiple threads. Equation 5.1 and 5.2 show the calculation and result when 4 threads are used. The actual results, in table 5.5, show a difference of 0.35, for the 64 bit searches and -0.29, for the 8 bit searches, on 4 threads with disabled hyper-threading.

$$\text{Speedup}_{enhanced}\left(0.99, 4\right) = \frac{1}{(1-0.99)+\frac{0.99}{4}} = 3.88 \tag{5.1}$$

$$\text{Speedup}_{enhanced}\left(0.875, 4\right) = \frac{1}{(1-0.875)+\frac{0.875}{4}} = 2.91 \tag{5.2}$$

Table 5.5 and the theoretical results from Amdahl's law show that libssa scales good to multiple threads, especially when considering that the used CPU implements the turbo boost technique. The executions on 1 thread are most likely done on a higher clock rate, which reduces the improvement of multiple threads.

Concluding one can say that for a most stable performance, hyper-threading should be disabled. If it is enabled then all possible hyper-threads should be used, to get the best performance.

5.1.5 Chunk sizes

One of the parameters that can be set in libssa is the size of the database chunks. These are processed by the threads, one at a time (see chapter 3.3.1). To evaluate the influence of this parameter on the performance,

[4]https://perf.wiki.kernel.org/index.php/Main_Page, last visited 23.4.2015

a benchmark is run using a number of constant test configurations with different chunk size configurations. The goal of it is to find a range of chunk sizes where libssa performs best.

Configurations The number of configurations tested against the chunk sizes include two databases of different sizes and average sequence lengths (UniProtKB/Swiss-Prot and Rfam), as well as two different thread configurations (1 thread and 8 threads). For each database, a sequence of about average length is chosen and the database searches are done using the 8 bit Smith-Waterman and Needleman-Wunsch algorithms. The query sequences are *P19930* for the Uniprot database and *5S_rRNA* for the Rfam database. Both were chosen as they are of about average length compared to their databases. Each of these configurations is run with 11 different chunk sizes: 10, 100, 500, 1 000, 1 500, 2 500, 5 000, 10 000, 25 000, 50 000, 100 000.

Results Figure 5.10 shows the results of the benchmark for both databases in both thread configurations. Both axes of the charts are shown log scaled. The lowest average runtime when using 1 thread is at a chunk size of 5 000 sequences for the UniProt database and at a chunk size of 1 500 sequences for the Rfam database. When using 8 threads, the lowest average runtime is at a chunk size of 500 sequences for the Rfam database, and 5 000 sequences, for the UniProtKB/Swiss-Prot database. Table 5.6 shows the exact values for the lowest runtimes.

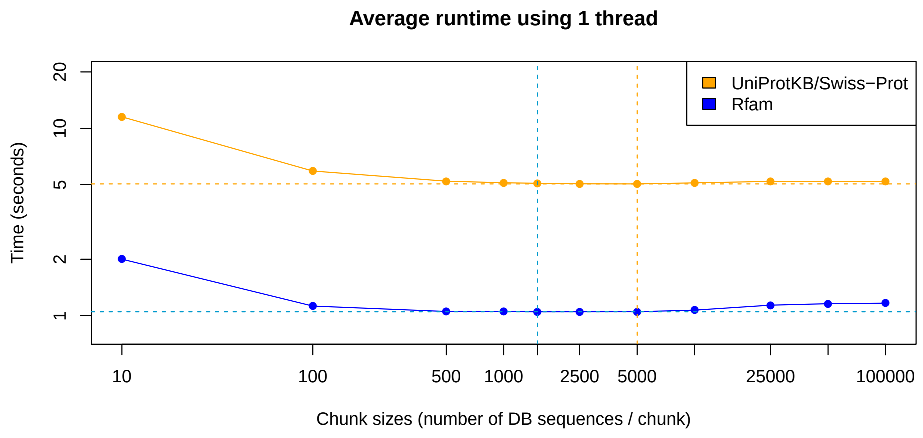
	1 thread	8 threads
Rfam	1.372 s	0.389 s
UniProtKB/Swiss-Prot	5.033 s	1.45 s

Table 5.6: Lowest average runtimes per configuration

Discussion Both charts in figure 5.10 show a performance drop to the lower end, of the chunk sizes. This has two reasons. First of all, a chunk size of 10 fills only 10 of the 32 available channels, reducing the effect of the computation over multiple channels. Secondly, the initialisation of a new chunk contains some processing overhead. This reduces the performance, if it is done too often.

The second performance drop, shown for the test run using 8 threads, occurs due to too big chunk sizes. These are so big that some threads are doing nothing. The Rfam database, for example, contains only 383 004 sequences. This means, when using 8 threads and a chunk size of 100 000 only 4 threads are working on a chunk, and 4 threads are waiting.

When only 1 thread is used, the chunk sizes have no upper limit, except one set by the hardware. Although if the chunk size becomes bigger than the size of the database, more memory is allocated than needed. This does



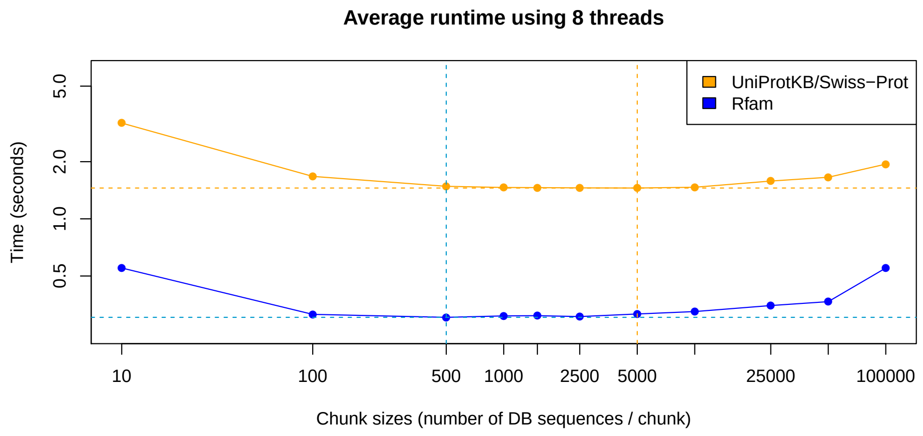


Figure 5.10: Average runtime of libssa, per chunk size, using different thread and database configurations. The dotted lines mark the shortest runtimes per database configuration.

not necessarily affect the performance of libssa but should be considered when using libssa in combination with other programs.

The optimal range of chunk sizes for both thread and database configurations is from 500 to 5 000 sequences. Table 5.7 shows the absolute difference between the lowest and highest runtimes, in that range. The default chunk size in libssa of 1000 sequences was chosen to be inside this range of high performance. Furthermore it supports smaller databases and ensures a good distribution of the workload to the threads.

	1 thread	8 threads
Rfam	0.0039 s	0.0126 s
UniProtKB/Swiss-Prot	0.1764 s	0.0305 s

Table 5.7: Absolute differences between the lowest and highest runtimes, using chunk sizes of 500 to 5 000 sequences.

5.1.6 Comparison of alignment tools

To get a better picture of the performance of libssa, the runtime is compared to different database search tools. For a better comparison, only tools were chosen, which implement the database searches based on optimal alignments using SIMD vectors. These tools are SWIPE, Opal, and SSW, with SWIPE and Opal implementing Rognes' approach and SSW implementing Farrar's approach (see chapter 2.3.4 and 2.4.3).

The comparison of Needleman-Wunsch database searches is only done between libssa and Opal, as SSW and SWIPE do not implement Needleman-Wunsch searches.

Configuration The focus of this test run is an evaluation of the runtime performance of libssa in comparison to similar database search tools. Therefore a configuration is chosen, which can be set in all compared tools. These are then run in the lowest available bit width and in all available SIMD configurations against four query sequences of varying length. The base configuration is the following:

- Database: UniProtKB/Swiss-Prot
- Gap open and extension costs: -3 and -1
- Substitution matrix: BLOSUM50
- query sequences: *O74807*, *P19930*, *Q3ZAI3*, and *P18080*

The gap costs of -3 and -1 for opening and extending a gap are chosen as they were already used by the developers of Opal for comparing sequence alignment tools. However, these should not have an influence on the performance of the tested tools.

For each program the source code was downloaded and compiled on the test system using the provided make files. The source code was downloaded on 8 April 2015.

Results Figure 5.11 and 5.12 show the runtimes of the different alignment tools per query sequence for the Smith-Waterman and Needleman-Wunsch algorithms. Both axes in the charts are log scaled. All of the curves exhibit a similar behaviour, the runtime increases with growing sequence lengths.

Additionally, table 5.8 and 5.9 show the average improvements of libssa over the competing tools for both algorithms. These are calculated by dividing the average runtime of the compared tool by the runtime of libssa.

Discussion In both charts one can see that libssa performs better than the competing alignment tools. For the Smith-Waterman it is at least 1.38 times faster than Swipe, the next fastest tool. Using the Needleman-Wunsch algorithm, the SSE version of libssa is as fast as the AVX version of Opal.

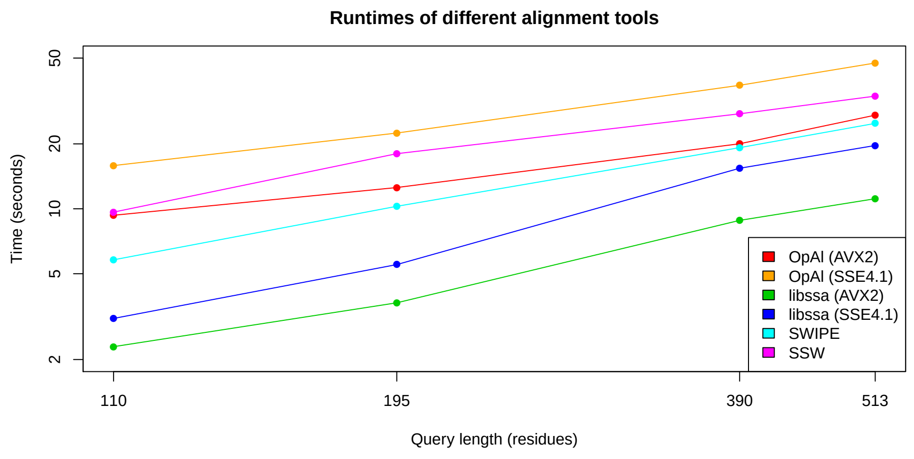


Figure 5.11: Chart displaying the runtimes per query length of different alignment tools using the Smith-Waterman algorithm.

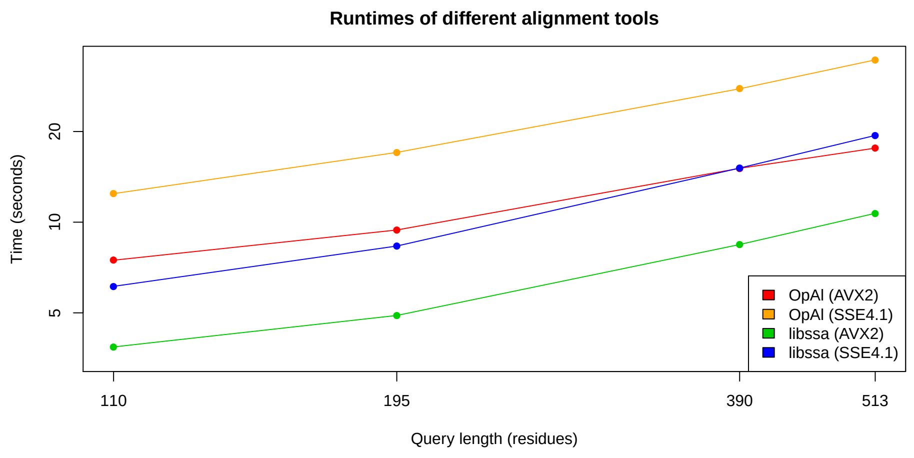


Figure 5.12: Chart displaying the runtimes per query length of different alignment tools using the Needleman-Wunsch algorithm.

	libssa (SSE)	libssa (AVX)
Opal (SSE)	**2.82**	4.75
Opal (AVX)	1.58	**2.66**
SSW	**2.03**	3.41
SWIPE	**1.38**	2.32

Table 5.8: Ratio of improvement of libssa over comparable tools using the Smith-Waterman. The values in bold face mark the ratios between tools of the same SIMD configuration.

Comparing the SSE and the AVX versions to each other, one can see that libssa is at least 1.78 times as fast as Opal.

	libssa (SSE)	libssa (AVX)
Opal (SSE)	**1.87**	3.29
Opal (AVX)	1.01	**1.78**

Table 5.9: Ratio of improvement of libssa over comparable tools using the Needleman-Wunsch. The values in bold face mark the ratios between tools of the same SIMD configuration.

SSW implements Farrar's approach [Farrar, 2007], which was already proven to be slower by Rognes when he proposed his approach in 2011. Libssa confirms these results.

Opal and SWIPE implement Rognes' approach from 2011 like libssa. Swipe and Libssa are implemented in C, while Opal is implemented in C++. Since all are implemented in an equally fast programming language, the differences in the runtime are most likely due to differences in the implementation.

In SWIPE, the core cell computation is implemented using manually written inline assembly, while libssa and Opal use Intel's intrinsics functions here. These can be better optimised by the compiler than manually written assembly. Another inefficient part of SWIPE is the overflow handling. These are detected after a chunk was searched through, while in libssa these are detected on the first search block they occur on. Affected sequences are then marked as overflown and exchanged with the next sequences.

The main difference to Opal is the size of the search blocks. Opal implements a size of 1, while libssa uses a size of 4. Wider search blocks result in less executions of the checks for overflows and finished sequences. Additionally Opal implements only 1 state, meaning the initialisations and checking for finished sequences are done every time. Furthermore Opal implements less parts of the algorithm using SIMD operations. The most important of them are the filling of the score profile and the overflow checks, which are done sequentially instead of being performed on SIMD registers.

Two conclusions can be drawn from this comparison. First of all the wider AVX registers speed up the execution of the database searches independent from the implementation. If the implementation is equal for both parts, the part done on AVX registers is faster. Secondly one can see that libssa is more efficient than competing tools, implementing optimal alignments for database searches. The increase in efficiency is based on more vectorisation and a more advanced algorithm.

5.1.7 Tests

An important part of modern software development are tests. They give confidence in the software working correctly and in that all use cases are implemented. This part discusses the test suite that was implemented for libssa. First an overview of the code coverage of the test suite is given, followed by a discussion of meaningful tests and a discussion of a common mistake in test development.

The code coverage is a measurement for the completeness of tests. It measures the percentage of lines in the source code, that are covered by at least one test case. In libssa the code coverage is measured using the tool gcov, integrated in GCC. This tool instruments the source code and collects information on the execution of code segments. This is done during the execution of the test suite. Afterwards an HTML report is generated using the tool lcov.

The current average code coverage for all source code files in the `src` directory is 98.4%, while 99.2% of all functions are covered by tests. These tests include positive and negative tests, to check as many lines as possible.

Some of the untested lines contain instructions to abort the program, in case of a programmer's error. These could happen, if, for example, a module is used that was not initialised before. This error occurs when a programmer does not follow the rules defined by a module or function. These should be visible as soon as possible so the programmer can fix these. They are intentionally left untested, except for the function for terminating the program (`fatal` in `util.c`) was tested.

An important part of software testing is finding meaningful tests. The problem is, when a test suite is only build for statement coverage, it does not test the correctness of the implemented algorithms. Even when it has a code coverage of 100%. In libssa most of the tests check the results of the algorithms as well. In case of the sequence alignments, these results are the computed alignment scores and the CIGAR strings. For a set of short sequences, these results were confirmed by hand. For longer sequences they are confirmed by the other implementations of the same algorithm in a different bit width. The hypothesis here is that it is less likely for an error to be present in both implementations.

The optimal method of testing would be to have external testers not known to the implementation, but with a good knowledge of the algorithms. Most likely they would come up with test cases that were not thought of during the development of libssa. This should be improved in the future. Either by finding external testers or by spending more time on finding test cases.

A common mistake in testing is to call each tested function only once per test. The test then checks that this function can be called exactly once. Its behaviour when it is called twice or more times, remains untested. A bug like this happened during the development of libssa in the `thread_pool` module. In each test it was initialised with a number of threads that were used, and afterwards the module was terminated. Later, when the benchmark of the multi-threading performance was executed, a segmentation fault occurred. At this point, the benchmark had changed the number of threads from 1 to 2. The `thread_pool` module was terminated and initialised with the new number of threads. The problem was that internally the memory used for the threads was released, but the variable was not reset to 0. When the module was initialised again, it checked the variable for 0, noticed that it was not 0, and proceeded to use it. Here the segmentation fault occurred, because memory was used that was not allocated.

Here one can see the importance of a well designed test suite and that a program can still have bugs, even when all tests run successfully. This makes tests at system and acceptance level even more important. These usually test a program from a different perspective, finding errors the programmer did not see.

5.2 Evaluating optimisations

This chapter presents a couple of optimisations, that were done during the development of the library. It covers optimisations of the algorithms as well as the implementation. Chapter 5.2.1 focuses on the algorithm while chapter 5.2.2 and 5.2.3 focus on the efficient implementation.

5.2.1 Search columns

A main feature of the vectorised database searches is the column-wise alignment of database sequence to the query sequence. The implementation follows Rognes [2011] approach using columns that are four residues wide with each column being aligned in one of two states: one where new sequences start, and one where only previous sequences are continued (see chapters 2.2.3).

During the implementation of an algorithm, one often has to decide, whether the implementation should be fast or easily readable and maintainable. When comparing the implementations for both states, one notices that each state is implemented in its own function and the main loop is divided into both states (see the files `search_simd_nw.c` and `search_simd_sw.c` and the chapters 3.3.3 and 4.4.2). Here one state is more complex by additionally saving the scores of the finished sequences and loading and initialising new sequences, while the other state computes only the alignment. Removing the less complex state would give the same results and optimise the readability and maintainability, but it would also increase the number of instructions carried out, when no new sequences start.

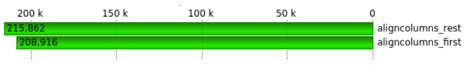


Figure 5.13: 8 bit AVX2 search (32 channels) on the Rfam database

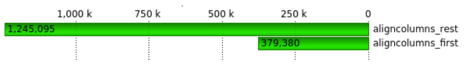


Figure 5.14: 8 bit AVX2 search (32 channels) on the UniProtKB/Swiss-Prot database

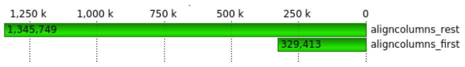


Figure 5.15: 16 bit SSE search (8 channels) on the Rfam database

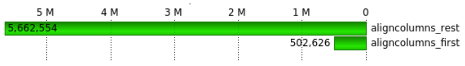


Figure 5.16: 16 bit SSE search (8 channels) on the UniProtKB/Swiss-Prot database

One criterion for deciding whether to keep both states or favour a simpler implementation with only one state is the number of executions of each state. If the simpler and possibly faster state is executed a lot less than the more complex state, an implementation with only one state might be favourable.

Table A.1 in appendix A presents the theoretical number of executions of both states. It shows the maximum number of executions of the state `no sequences ended` and the minimum number of executions of the state `sequences ended`. It presents these for the optimal case, when no padding is applied, and for the worst case, when each sequence is fully padded by 3 symbols. The number of columns is calculated by dividing the total number of residues by the number of channel, while a block column packs four columns. These resemble the columns searched by the algorithm for one search block. They can either be computed in the state `no sequences ended` or `sequences ended`. In the worst case, each sequence starts at its own block column resulting in the shown number of executions of the state `sequences ended`. The number of executions for the state `no sequences ended` is the difference between the number of block columns and the number of executions of the other state.

The differences, Table A.1, between the Rfam and the Swiss-Prot database are the number of sequences, the total number of residues, and the average sequence lengths: 138 residues in the Rfam database and 356 residues in the Swiss-Prot database.

Figure 5.13 to 5.16 illustrate another approach for deciding whether to implement two states. They illustrate the actual numbers of executions of both states. These are presented for database searches using local alignments on the Rfam and the UniProtKB/Swiss-Prot database with the maximum and minimum number of channels. The function `aligncolumns_first` implements the state with new sequences while `aligncolumns_rest` implements the other one. The charts in figure 5.13 to 5.16 are created using the program VampirTrace[5] for tracing the executions and Vampir[6] for displaying the results.

Comparing table A.1 and figure 5.13 to 5.16 one notices that the more complex state `aligncolumns_first` is always executed less times than its theoretical maximum. At the same time the simpler state is executed more often. Furthermore the difference between the number of calls in both states is less when more channels are used. Taking the average number of residues (138 in Rfam and 356 in Swiss-Prot) into consideration, one can see that using two states becomes more effective for databases with longer sequences.

Libssa is implemented using two states. These form a core part of the database searches, hence partially performing these in a simpler state, using less instructions, is expected to speed up the computation. The comparison to Opal partially confirms this, as it is implemented using only one state. Although it is not a complete confirmation since this is not the only difference to libssa, as shown in chapter 5.1.6.

5.2.2 Memory management

Memory management is an important part of software development. One has to make sure that no memory leaks occur and that the memory is used efficiently. In C and C++, the memory used for data can be divided into three different parts[7]:

- Static memory: This is used for variables defined outside of functions which are declared using the `static` keyword. The life cycle of these variables is the whole runtime of the program.
- Automatic memory: This is used for variables declared inside of functions without the static keyword. These are put on the stack, their lifetime is the runtime of the function.
- Dynamic memory: This memory is located on the heap and can be allocated using one of the `malloc` functions. This memory is valid until a call to `free` deallocates it.

When deciding in which space to place a variable, the important factors are the lifetime and the size of the data. Memory on the stack is fast, but its size is finite. The heap on the other hand contains usually more memory, but its allocation and deallocations causes some overhead. Another problem, especially with heap memory, are memory leaks. These occur, when memory is allocated, but never deallocated. Tools like `valgrind` help finding these,

[5]http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace, last visited 20.3.2015

[6]https://www.vampir.eu/, last visited 20.3.2015

[7]http://www.design-reuse.com/articles/25090/dynamic-memory-allocation-fragmentation-c.html, last visited 21.3.2015

however preventing them is the task of the developer.

The following paragraphs describe some improvements that were made in memory management and their effects. The main concept, behind these improvements was to re-use memory instead of allocating new memory and to keep short living objects on the stack.

Storage of sequences in chunks Each database sequence, that is searched for or aligned to a query sequence, is converted from ASCII format to an integer representation. See chapter 4.1.2 for more details on this.

In a first version of the `db_adapter` module new memory was allocated for each database sequence. The mapped sequence and some additional information were stored in it, the data was used in the database search, and the memory was freed afterwards. Additionally memory was allocated once for the chunk object and the list pointers to the sequence data. As an example 1 + 1 + 100 000 + 100 000 = 200 002 calls to `malloc` were done for a database of 100 000 sequences. One for allocating the chunk struct, one for allocating the array of sequence data pointers, and two for each sequence data struct and the mapped sequence.

The current version, of the `db_adapter` module allocates memory for each chunk and each sequence in it once, stores the sequence data in it, and re-uses it for the next chunk. For a database of 100 000 sequences and a chunk size of 1 000, this results in 1 + 1 + 1 000 + 1 000 = 2 002 allocations. One for allocating the chunk struct, one for allocating the array of sequence data pointers and two for each sequence data struct and the mapped sequence of the chunk. Since every sequence has a different size, a call to `realloc` is done for the mapped sequence memory for each database sequence. This adds 100 000 calls to `realloc` for the example.

Even if calls to `realloc` are as slow as calls to `malloc`, the improved version uses 98 000 allocations (49%) less than the previous version. In a next step, the number of reallocations could be reduced by allocating enough memory for each sequence to hold the longest sequence in the memory. This way no reallocations need to be done. The drawback is that more memory than needed would be used.

Storage of new elements in the result heap The results of a database search are kept in a priority queue (see chapter 4.3.2). In the first version, the memory for a new element was allocated on the heap. The new element was then initialised, passed to the queue which stored a copy of the element, and the elements memory was freed afterwards. The result of it were allocations of heap memory in the number of searched database sequences.

The current implementation, allocates the new elements on the stack, instead of the heap. This way no calls to `malloc` and `free` are done. The differences in the runtime are shown in figure 5.17. On the left the data is shown before and on the right after the changes. The function `add_to_minheap` contains the allocation, initialisation, and freeing of the elements on the heap. The difference in runtime, for the function `xrealloc` might be due to less calls to `malloc`. Allocating, the new elements for the result queue on the stack might have reduced the overhead for allocating and reallocating memory on the heap. This could speed up dynamic memory allocations outside the scope of the function `add_to_minheap`.

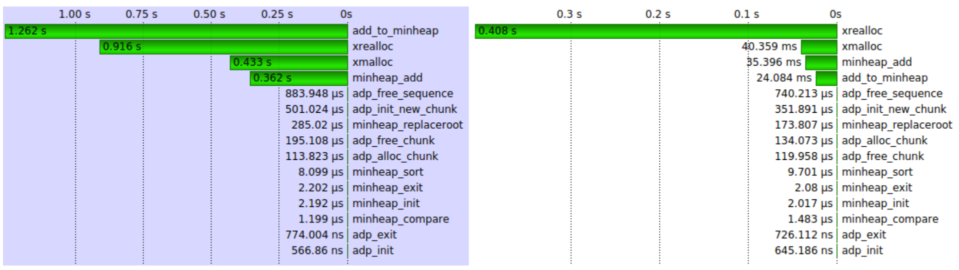


Figure 5.17: Accumulated exclusive runtime of selected functions. On the left side are the runtimes before and the right side after the changes.

Storage of elements in the overflow lists A third part that was improved in terms of memory management, are the overflow lists (see chapter 4.4.2). In the first version, these were implemented as a linked list, where new items are added to the end when an overflow was detected. This linked list was then converted to a chunk and re-computed with a higher bit width. Elements added to the list were allocated on the heap.

In database searches, where only a few overflows occur, this implementation is memory efficient as it does not preallocate memory that is not used later on. However, in some cases overflows occur in greater numbers, as one can see in table 5.1 on page 65. The current implementation ensures a more constant performance by allocating a second database chunk that is filled when overflows are detected. The overhead for allocating this chunk is small as it only consists of two allocations: one for the data structure and one for the list of pointers to database sequences. This list is filled during the search with the pointers to database sequences where an overflow occurred. This way, the overflow list does not need to be converted, as it is already a chunk data structure, which can be processed by the database search functions.

5.2.3 Optimisations done by the compiler

Modern compilers are good at writing optimised machine code. Often it is enough to implement the software in a readable and maintainable way, and the optimisation is done by the compiler. The `dprofile_fill` functions which compute the temporary score profiles (see chapter 4.4.2) are an example where optimisations can be left to the compiler, yet they also show the limits of it.

Listing 5.1 shows a part of the `dprofile_fill_8_sse41` function, for creating the score profile for the 8 bit SSE search. By using constants like `CHANNELS_8_BIT` and `CDEPTH_8_BIT`, the compiler was able to roll out all of the loops, omitting the instructions used for the loops. In the same step, most of the calculations, like $x + 1$, could be changed to the actual numbers, based on the constants. Unrolling these loops manually would make the code more prone to errors and harder to read and maintain.

Listing 5.1: Part of dprofile_fill_8_sse41

```
for( int x = 0; x < CHANNELS_8_BIT; x++ ) {
    xmm[x] = _mm_load_si128( (__m128i *) (score_matrix_8 + d.a[
        x] + i) );
}
// transpose matrix
for( int x = 0; x < CHANNELS_8_BIT; x += 2 ) {
    xmm_t[x + 0] = _mm_unpacklo_epi8( xmm[x + 0], xmm[x + 1] );
    xmm_t[x + 1] = _mm_unpackhi_epi8( xmm[x + 0], xmm[x + 1] );
}
...
// store matrix
for( int x = 0; x < CHANNELS_8_BIT; x++ ) {
    _mm_store_si128( (dprofile + CDEPTH_8_BIT * (i + x) + j),
        xmm[x] );
}
```

Listing 5.2 shows a loop the compiler could not completely optimise. This loop iterates over all 32 channels filling an integer array with the contents of the search block shifted by five. The integers in `d` are later used as indices to the score matrix.

Listing 5.2: Original source code for dprofile_fill_8_avx2

```
int d[CHANNELS_8_BIT];

for( int i = 0; i < CHANNELS_8_BIT; i++ )
    d[i] = dseq_search_window[j * CHANNELS_8_BIT + i] << 5;
```

Listing 5.3 shows the compiler optimised assembly code for the code in listing 5.2. It was generated using the GCC option `-S` along with `-O3` for best optimisation. The compiler manages to unroll and partially optimise the loop. It uses the 128 bit wide XMM registers to vectorise 4 times 8 iterations of the loop. 8 iterations fit into one registers, since the indices stored in `d` are maximally 16 bit wide and 8 of these can be processed in one SSE register.

A better optimisation is done manually, as shown in Listing 5.4. This version uses the 256 bit wide AVX registers to load 16 values instead of 8. This way the loop is processed in two iterations using less than half of the instructions of the compiler optimised version. The assembly code to the manually optimised loop is shown in listing 5.5.

Listing 5.3: Assembly for original code of dprofile_fill_8_avx2

```
.L19:
    vmovdqu         (%rsi,%r8), %xmm0
    xorl            %eax, %eax
    movq            score_matrix_8(%rip), %rcx
    vinserti128     0x1, 16(%rsi,%r8), %ymm0, %ymm0
    vpmovzxbw       %xmm0, %ymm1
    vextracti128    0x1, %ymm0, %xmm0
    vpmovzxwd       %xmm1, %ymm2
    vextracti128    0x1, %ymm1, %xmm1
    vpmovzxbw       %xmm0, %ymm0
    vpslld          $5, %ymm2, %ymm2
    vmovdqa         %ymm2, -2192(%rbp)
    vpmovzxwd       %xmm1, %ymm1
    vpslld          $5, %ymm1, %ymm1
    vmovdqa         %ymm1, -2160(%rbp)
    vpmovzxwd       %xmm0, %ymm1
    vextracti128    0x1, %ymm0, %xmm0
    vpslld          $5, %ymm1, %ymm1
    vmovdqa         %ymm1, -2128(%rbp)
    vpmovzxwd       %xmm0, %ymm0
    vpslld          $5, %ymm0, %ymm0
    vmovdqa         %ymm0, -2096(%rbp)
    .p2align 4,,10
    .p2align 3
.L20:
```

Listing 5.4: Optimised source code for dprofile_fill_8_avx2

```
union {
    __m256i v[2];
    int16_t a[CHANNELS_8_BIT];
} d;

for( int i = 0; i < 2; ++i ) {
    __m256i tmp = _mm256_loadu_si256( (__m256i *) (
        dseq_search_window + (j * CHANNELS_8_BIT + i * (
        CHANNELS_8_BIT / 2))) );

    _mm256_store_si256( &d.v[i], _mm256_slli_epi16( tmp, 5 ) );
}
```

Listing 5.5: Assembly for optimised code of dprofile_fill_8_avx2

```
.L18:
    vmovdqu         (%rsi), %ymm0
    movq            score_matrix_8(%rip), %r8
    xorl            %eax, %eax
    vpsllw          $5, %ymm0, %ymm0
    vmovdqa         %ymm0, -2128(%rbp)
    vmovdqu         32(%rsi), %ymm0
    vpsllw          $5, %ymm0, %ymm0
    vmovdqa         %ymm0, -2096(%rbp)
    .p2align 4,,10
    .p2align 3
.L16:
```

A fast method for finding optimisation possibilities, like shown above, is the GCC option `-fopt-info-missed`. This option prints information on missed optimisation opportunities, like the following:

```
search_8_util.c:239:41: note: Failed to SLP the basic block.
search_8_util.c:239:41: note: not vectorized: failed to find SLP
    opportunities in basic block.
```

SLP (Superlevel Word Parallelism) is an approach to parallelism looking for vectorisation opportunities in straight line code. GCC uses this method to find loops that can be vectorised. [Rosen et al., 2007]

5.3 Exploiting open source

One of the advantages of open source software is that it can be re-used in other software. However, some of the open source licenses require the new software to be released as open source as well. Additionally, the external source code should be marked as taken from another project. This chapter discusses the source code taken from other open source projects.

The source code re-used in libssa is mostly released under the AGPL v3 license[8] and part of it under the MIT license[9]. The MIT license is a permissive non-copy-left license, meaning source code released under it can be used in any other project. The APGL v3 on the other hand is a copy-left license, requiring the new software to be released as open source as well. Libssa complies to this, as it is released under the AGPL v3. The source code taken from other projects is marked with a comment in the source code.

The parts of libssa that are based on other projects are the reading of query sequences, the internal format and translations, and parts of the database searches and alignments. All were taken from the following projects: SWIPE, SWARM, VSEARCH, and SSW. These are all released under the AGPL v3, except SSW, which is released under the MIT license.

The reading of query sequences is based on the implementation in SWIPE. Reading the FASTA sequence from a file and converting it to the internal format is taken from there, while libssa additionally reads in sequences from a string.

The idea of the internal format is implemented in SWIPE, SWARM, and VSEARCH. Libssa re-uses the implementation from SWIPE and generalises it. In libssa, the conversion to the internal format is implemented in the module `util_sequence`, which does the conversions for database and query sequences. In the other programs, this conversion is implemented

[8]http://www.gnu.org/licenses/agpl.html, last visited 20.4.2015
[9]http://opensource.org/licenses/MIT, last visited 20.4.2015

twice, for the query and for the database sequences.

The translation of nucleotide to amino acid sequences is implemented in the `util_sequence` module as well. This part is also taken from SWIPE and generalised the same way as the conversion to the internal format. All functions were changed to increase the readability. They work now on sequence structures instead of pairs of a string and a length parameter.

For the database searches, the 64 bit Smith-Waterman implementation is taken from SWIPE, while the 64 bit Needleman-Wunsch implementation is derived from the Needleman-Wunsch-Sellers implementation in SWARM. The concept for the vectorised database searches is based the vectorised 16 bit Needleman-Wunsch implementation in VSEARCH. This algorithm was first reduced and optimised before an 8 bit version and the 8 and 16 bit Smith-Waterman implementations were derived from it. In VSEARCH, the database search is coupled in a much tighter way with the database. Here the search algorithm is initialised with a list of sequence numbers and the algorithm retrieves the sequences directly from the database. These are stored in the internal format. In libssa the alignment algorithms do not have any knowledge of where the sequences come from. They work on chunks of sequences only, provided by the outer layer which itself has no knowledge on the alignment algorithms. Another part that was changed is the computation of the CIGAR string, which is done during the vectorised global alignment in VSEARCH. In libssa, the computation of the CIGAR strings is done in a separate module.

The computation of the score profile is also taken from VSEARCH. Although SWIPE and SWARM implement similar approaches. The changes in libssa include adapting it to the AVX instruction set and generalising it. In libssa, 4 different versions are implemented for the different bit widths and instruction sets, while all 4 versions follow the same design. Also the design of using loops with a constant number of iterations is new in libssa.

The min-max-heap implementation in libssa is the one from VSEARCH. Here the structure for an element of the heap was changed and a set of tests was added.

The computation of the alignment is based on the implementations in SWIPE and SSW. The changes here are mostly adapting the code to libssa and modularising it. The part from SSW was taken since it is more clearly implemented, although SWIPE implements a similar algorithm.

SWIPE, SWARM, and VSEARCH use a similar concept, for splitting the workload on multiple threads like libssa. They all implement a concept of dividing the total workload into chunks, that are processed by worker threads. The difference to libssa is the encapsulation of the threads. In libssa, the `thread_pool` module wraps the calls to the `pthread` API and encapsulates the functions for managing the threads. This follows the modular design of libssa and makes the source code more readable and maintainable.

Another difference in the multi-threading implementations is the collection of results from the database searches. Libssa omits the synchronisation by maintaining local result lists per thread. These are returned to the main thread, which aggregates them to a global result list. In SWIPE, for example, results are collected in the main thread. When a worker thread reports a result, it acquires a lock on the result list, adds the new result, and releases the lock. Using a lock synchronises the worker threads and introduces time spend waiting. A lock-free implementation like in libssa is usually faster.

All source code taken from other projects had to be adapted to the design of libssa. Often it was implemented in a non modular way, where the different tasks were woven into each other, making it hard to replace or optimise certain parts. Another changed aspect are custom type definitions like `BYTE`, `WORD`, and `CELL` for integers in different bit width. Libssa uses the standard data types defined in `stdint.h`, like `int8_t`. For these types, the name already tells the size and signedness, making the source code easier to read, especially for developers who are not familiar with the project. Here, for example, one immediately knows the bit width of a type and what can be calculated using it.

For some implementations, from other projects, the main effort was to understand the source code and to make it more readable and maintainable. Sometimes it is easier to re-implement a part, like the handling of multiple threads, instead of reusing other code. The re-implemented part will immediately fit into the design of ones code, while external code always needs to be adapted.

Chapter 6

Future work

6.1 New and improved algorithms

Libssa is developed to add new algorithms. Here one only has to exchange an existing implementation or add a new algorithm to the modular structure.

6.1.1 Needleman-Wunsch-Sellers algorithm

In the current version libssa implements two optimal alignment algorithms: the Needleman-Wunsch and the Smith-Waterman. Another algorithm that is used is Sellers' version of the Needleman-Wunsch algorithm, which is for example used in SWARM. By implementing this algorithm in libssa the database search parts of SWARM could be replaced with libssa, potentially speeding up the computations in SWARM.

Chapter 2.2.3 describes the algorithm. An implementation of it would be mostly similar to the Needleman-Wunsch implementations in libssa with an exception to the score system which would be exchanged by Sellers' score system.

6.1.2 Different gap penalties

The Needleman-Wunsch algorithm in VSEARCH uses different gap penalties at both ends of both sequences and inside the alignment. This can be used to compute semi-global alignments.

Libssa applies in the current version the same gap costs to all parts of the alignment. Adding different gap costs for different part would make libssa useful in possibly more projects.

6.1.3 Optimising alignments

The computation of CIGAR strings is not optimised in libssa. It is done non vectorised on 64 bit and the whole direction matrix is kept in memory. For long sequences this can become a performance issue due to the quadratic memory requirements.

Myers and Miller [1988] described an algorithm computing the CIGAR string with linear memory consumption. The algorithm is already implemented in SWIPE and VSEARCH from where it could be adapted to libssa.

Here one has to decide whether to always use the linear memory aligner. It does have lower memory requirements, but it also has a higher runtime. Another option would be to implement an algorithm deciding when to use it, for example when the sequences become to long for the non linear aligner. One could also add a parameter to the API for the user to decide which alignment algorithm to use.

Another potential optimisation of the alignments is doing them on SIMD vectors. Here one could re-use the vectorised database searches and add instructions for gathering the direction matrix. This matrix is then used to compute the CIGAR string of the alignment. Computing the alignments on SIMD vectors would speed up the computation of CIGAR strings if a high number of alignments is to be computed.

6.2 Error handling

A good error handling strategy is to validate user input and assert programmer input. If a programmer writes code violating the constraints set by a program an error should be raised that cannot be overseen. This is done with assertions which abort a program if they fail. At the same time, input from a user should be validated to provide a meaningful error message and to prevent an abortion of the program.[1]

Libssa implements this approach partially. The user input is validated and error messages are shown. Also programmer's input is asserted. The part that is not yet optimally implemented is when the allocation of memory fails. Here the program terminates, while it should return gracefully cleaning up all previously allocated and initialised data.

Here one could use the functions `setjmp`[2] and `longjmp`[3] of the header file `setjmp.h`. These can be used to implement a basic try and catch for C. If this is used on different levels, each level could catch errors, clean up all used data, and then return with a meaningful error message. The implementation of the try and catch can be done using macros, which would keep the source code readable and allow for a consistent implementation throughout the library.

[1] https://programmers.stackexchange.com/questions/64926/should-a-method-validate-its-parameters, last visited 28.4.2015

[2] http://man7.org/linux/man-pages/man3/setjmp.3.html, last visited 28.4.2015

[3] http://man7.org/linux/man-pages/man3/longjmp.3.html, last visited 28.4.2015

The error messages are best implemented as enumerations. These provide more information than a simple integer constant, while being more efficient than directly returning the error messages. The user then asks the API for the error message to that enum.

A good error handling is defensive, while it still does not hide errors. Errors that should be fixed during the development of the program should abort the execution. When the program is used on the other hand an error should never terminate it. It should give feedback on what went wrong and how to avoid it.

6.3 Creating the temporary score profile

An important part of the database searches is the creation of the temporary score profile. For each version of the database searches (SSE/AVX and 8 bit/16 bit) a separate version for filling the score profile is implemented. Chapter 4.4.2 describes these and their integration in the search algorithms. This section proposes three possible improvements of these functions, to further reduce the runtimes.

The current implementation creates the score profile by selecting the matching lines of the score matrix, transposing these, and storing them in the score profile. Each version computes on the same bit width as the respective database search. Equation 6.1 shows the number of instructions for each of the implemented versions.

$$\begin{array}{lcccccccl}
 & & \text{Depth} & & & \text{Prepare indices} & & \text{Transpose} & \\
C_{8bitSSE} & = & 4 & * & (& 2*8 & + & 2*6*16 &) = 832 \\
C_{8bitAVX} & = & 4 & * & (& 2*10 & + & 7*32 &) = 976 \\
C_{16bitSSE} & = & 4 & * & (& 7 & + & 4*5*8 &) = 668 \\
C_{16bitAVX} & = & 4 & * & (& 8 & + & 2*6*16 &) = 800
\end{array} \tag{6.1}$$

6.3.1 Compute on 8 bit

All values of the substitution matrices are inside the 8 bit range. Common scores, such as the ones of the BLOSUM and PAM matrices are in the range of -20 to $+20$. Therefore it would be possible to load and transpose the scores in 8 bit, and store them unpacked in 16 bit.

If the computations are done on 8 bit instead of 16 bit twice as many substitution scores are loaded into the registers. Hence also twice the amount of data is transposed in one step. This would reduce the number of transpose operations by a factor of 2, improving the performance of this part of the computation.

This way, only two versions for the transpose operation would be implemented: an 8 bit SSE and an 8 bit AVX version. The 16 bit implementations would use the 8 bit transpose implementations and afterwards unpack the transposed data to store it in 16 bit.

6.3.2 Different implementation for nucleotide sequences

During the database searches libssa does not distinguish between nucleotide and protein sequences. For nucleotide sequences which have only 16 different symbols this mean half of the score matrix and profile is padding. Only one half contains meaningful data.

A possible optimisation would be implementing separate versions of creating the score profile, for nucleotide and protein sequences. The nucleotide version would then compute on only the first 16 by 16 symbols of the score matrices, possibly reducing the amount of used instructions by half. Equation 6.2 shows the possible numbers of instructions for the improved version. Here one can see a possible reduction, of the instructions counts by a factor of 2. The calculations for the indices is not reduced in the improved version. Therefore are the instruction counts reduced by only almost a half.

$$\begin{array}{lccccccccc}
 & & \text{Depth} & & & \text{Prepare indices} & & \text{Transpose} & & \\
C_{8bitSSE} & = & 4 & * & (& 2*8 & + & 1*6*16 &) = & 448 \\
C_{8bitAVX} & = & 4 & * & (& 2*10 & + & 6*32 &) = & 528 \\
C_{16bitSSE} & = & 4 & * & (& 7 & + & 2*5*8 &) = & 348 \\
C_{16bitAVX} & = & 4 & * & (& 8 & + & 1*6*16 &) = & 416
\end{array} \tag{6.2}$$

6.3.3 Filling for only match/mismatch values

Another improvement is to compute the score profile different if constant match and mismatch scores are used. In this case the internally used score matrix is filled with the mismatch costs, except the main diagonal which contains the match scores.

In this case it could be faster to use `memset` to set the whole score profile to the mismatch scores and then change only these values where a match occurs. This would result in 128 set operations for a 5 residue wide search block with 32 channels, which is far less than the 976 instructions the current implementation uses.

6.4 Other work

The current version of libssa is written in C and optimised for systems running one or more CPUs on a local node. This could be improved to broaden the audience of libssa.

The first optimisation could be wrapper tools, making libssa available to programs implemented in other programming languages. These are implemented in the target language and wrap all calls to libssa, such that the program using libssa only needs to call the functions of the wrapper without writing any C code.

The second optimisation could be extending libssa, such that it can distribute the computations on multiple nodes of a cluster. One technique that could be used here is message passing interface (MPI), with Open MPI[4] as the implementation of it.

[4]https://www.open-mpi.org/

Chapter 7

Conclusion

In this thesis a new library for database searches using optimal sequence alignment algorithms was presented. The main feature of the library is its modular structure, which allows to easily integrate new algorithms and exchange existing implementations with new versions. Each module is implemented as efficient as possible to speed up the database searches.

Two algorithms were implemented as a proof-of-concept for an efficient implementation of optimal sequence alignments. These and the overall database search are based on SIMD instructions implementing Rognes' approach. Libssa increases the already good performance of SWIPE, the reference implementation of Rognes' approach.

Furthermore libssa was used to show an increase in performance when the database searches are performed using AVX instructions compared to a computation on SSE instructions. For this purpose two versions of each algorithm are implemented. The benefit is that libssa can accelerate database searches on systems implementing SSE only, as well as on systems implementing AVX.

In comparison to similar libraries and tools, libssa shows a significantly improved performance. It is more than twice as fast using local alignments and about 1.8 times as fast using global alignments.

Bibliography

M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29:996–1000, 1986. doi: 10.1145/6617.6621.

M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23:156–161, 2007. doi: 10.1093/bioinformatics/btl582.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982. doi: 10.1016/0022-2836(82)90398-9.

D. R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley Professional Computing Series. ADDISON WESLEY Publishing Company Incorporated, 1996. ISBN 9780201498417.

S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89:10915–10919, 1992. doi: 10.1073/pnas.89.22.10915.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012. ISBN 9870123838728.

M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. doi: 10.1109/MC.2008.209.

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi: 10.1145/360825.360861.

H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009. doi: 10.1093/bioinformatics/btp352.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2009. ISBN 9780132350884.

E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970. doi: 10.1016/0022-2836(70)90057-4.

T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12(1):221, 2011. doi: 10.1186/1471-2105-12-221.

T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16:699–706, 2000. doi: 10.1093/bioinformatics/16.8.699.

I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC. *Proceedings of the GCC Developers' Summit*, pages 131–142, 2007.

P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974. doi: 10.1137/0126070.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981. doi: 10.1016/0022-2836(81)90087-5.

W. K. Sung. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Mathematical and Computational Biology. Chapman & Hall/CRC Press, 2010. ISBN 9781420070330.

A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13:145–150, 1997.

M. Zhao, W.-P. Lee, E. P. Garrison, and G. T. Marth. SSW library: An SIMD Smith-Waterman C/C++ library for use in genomic applications. *PLOS ONE*, 8(12):e82138, 2013. doi: 10.1371/journal.pone.0082138.

Appendix A

Number of search columns for different databases

See table A.1 on the next page.

	Rfam version 11			UniProtKB/Swiss-Prot Release 2015_03		
Sequences	383 004	383 004	383 004	547 964	547 964	547 964
Total residues	52 830 109	52 830 109	52 830 109	195 014 757	195 014 757	195 014 75
Total residues (max padding)	53 979 121	53 979 121	53 979 121	196 658 649	196 658 649	196 658 64
Channels	32	16	8	32	16	8
Size of search block	4	4	4	4	4	4
Columns	1 650 941	3 301 882	6 603 764	6 094 211	12 188 422	24 376 845
Columns (max padding)	1 686 848	3 373 695	6 747 390	6 145 583	12 291 166	24 582 331
Block columns	412 735	825 471	1 650 941	1 523 553	3 047 106	6 094 211
Block columns (max padding)	421 712	843 424	1 686 848	1 536 396	3 072 792	6 145 583
Executions of state `no sequences ended`	29 731	442 467	1 267 937	975 589	2 499 142	5 546 247
Executions of state `sequences ended`	383 004	383 004	383 004	547 964	547 964	547 964
Executions of state `no sequences ended` (max padding)	38 708	460 420	1 303 844	988 432	2 524 828	5 597 619
Executions of state `sequences ended` (max padding)	383 004	383 004	383 004	547 964	547 964	547 964

Table A.1: The table shows the theoretical maximal number of executions of the state `no sequences ended` and the theoretical minimal number of executions of the state `sequences ended`. The numbers are shown for the Rfam database in version 11 and the UniProtKB/Swiss-Prot database Release 2015_03, for different channel numbers. The number of columns is calculated, by distribution the residues equally to all channels. Block columns are, four residues wide.

Appendix B

Reproducing performance results

The performance results as shown in chapter 5.1 can be reproduced using the a set of tools and scripts provided by libssa. These are located in the sub directory `benchmark` in the libssa project folder:

- `data`: Contains the test data
- `results`: All test result are written here.
- `src`: Contains the C files implementing the benchmarks.
- `scripts`: Contains the R scripts to evaluate the results.
- `aligner_comparison`: Contains shell scripts for running benchmarks on different alignment tools.
- `Makefile`: Makefile for building all benchmarks.

The benchmarks are first build using the target `all` of the Makefile. This produces a binary for each benchmark in the directory `benchmark`. These run without any parameters and produce a comma separated value (CSV) file, in the directory `results`. The name of this file contains the benchmarks name and the current date and time. This file is then evaluated using the provided R scripts. Before these can be used, the constants in the script `evaluate_config.r` should be changed to reflect the current system. These set among others the output folder for the generated diagrams.

The following benchmarks measuring the runtime of different configurations are implemented:

- `base_test_run`: measures selected configurations to get an overview of the performance
- `chunks`: measures different chunk configurations
- `queries`: measures different query configurations
- `threads`: measures different thread configurations

The directory `aligner_comparison` contains a benchmark comparing libssa to a set of other alignment tools. This benchmark generates the data shown in chapter 5.1.6. This directory contains a set of `run`-scripts wrapping the calls to the alignment tools, and two shell scripts for executing the benchmark. The script `runner.sh` runs a command *n* times, where *n* and the command are specified as parameters and the script `aligner_comparison.sh` executes the benchmark.

The sub folder `data` contains some of the test data used in the benchmarks. The UniProtKB/Swiss-Prot and the Rfam database are not added to this folder to keep the size of the GIT repository small. The UniProt and the Rfam database are available online.

The code coverage results are reproduced using the make target `test_coverage` of the makefile in the project folder. Before the target is run the compiler option `-coverage` is set using the variable `DEBUG_FLAGS`. This target then runs first the test suite and afterwards creates the code coverage report in the directory `coverage_data`. The file `index.html` in the sub folder `cov_tests` shows the code coverage report.

Appendix C

Installing and running libssa

The source code for libssa is hosted on Github: https://github.com/RonnySoak/libssa. From there it can be downloaded and used on any Linux system. It was developed and tested on a 64 bit Linux Mint with a Kernel version 3.13.0. However, it should run on any recent 32/64 bit Linux.

Build is libssa using the `Makefile` in the project folder. The target `all` compiles libssa, the tests, and an example application. The requirements for this are an installation of GCC and a suitable database library. The requirements for the database library are described in chapter 4.2 and it is configured in the `Makefile` in the variables `DATABASE_LIB`, `DATABASE_LIB_FILE`, and `DATABASE_LIB_FOLDER`. For the development GCC was used in the version 4.9.2, but any GCC of at least version 4 should compile libssa.

When the compilation was successful the target `check` can be used to run the test suite. If all is set up correctly it should run without errors. Some of these test might fail if the CPU does not implement the SSE2, SSE4.1, and AVX2 instruction sets. These test cases can be identified by their names which have "sse" or "avx" as part of the name.

Libsdb can be used as a reference implementation of a database library. This library was developed alongside libssa and is available on Github as well: https://github.com/RonnySoak/libsdb. Libsdb is licensed under the AGPL v3, like libssa.

The folder `tests/testdata` contains some FASTA sequences and databases, which can be used to test out the library. For this an example program is provided in the `src` folder. This is compiled with the target `libssa_example` and the binary is found in the project folder. The example application is configured using a set of command line parameters. A call to `./libssa_example` gives an overview on these.