Dissertation zur Erlangung des Doktorgrades
der Fakultät für Chemie und Pharmazie
der Ludwig–Maximilians–Universität München

MMseqs: ultra fast and sensitive clustering and search of large protein sequence databases

Maria Hauser, geb. Piskareva-Vassilieva
aus St.-Petersburg, Russland

2014

Erklärung:

Diese Dissertation wurde im Sinne von §7 der Promotionsordnung vom 28. November 2011 von Herrn Dr. Johannes Söding betreut.

Eidesstattliche Versicherung:

Diese Dissertation wurde selbstständig und ohne unerlaubte Hilfe erarbeitet.

München, am 29. Juli 2014

Dissertation eingereicht am: 30. Juli 2014

1. Gutachter: Dr. Johannes Söding
2. Gutachter: Dr. Peer Kröger

Mündliche Prüfung am: 25. September 2014

Acknowledgements

I would like to thank all the people who made this work possible with all my heart.

A big thank you goes to my advisor Dr. Johannes Söding for supporting me through all the last years, for the possibility he gave me to contrubute to such a fascinating research project and for the great discussions. We also had some tough times and I am very happy that we managed to make it through together and emerge even stronger as a team.

My deepest gratitude and all my love to my mother, who always supported me in all respects, encouraged me, helped me to care for my son and was always there when I needed her. She and I are the greatest team.

Я хочу выразить огромную благодарность и любовь моей маме, которая помогала мне смотреть за моим сыном и всегда была около меня, когда я нуждалась в поддержке и помощи. Мы с тобой самая лучшая команда.

I would like to thank Martin for his enthousiasm for the computer science and the research which was always very inspiring for me, for his constant support and fruitful discussions.

I am very grateful to Dr. Peer Kröger for being my second PhD examiner and to Prof. Dr. Klaus Förstemann, Prof. Dr. Volker Heun, Prof. Dr. Roland Beckmann, Prof. Dr. Mario Halic and Dr. Dietmar Martin for offering their time as members of my dissertation committee.

I deeply appreciate the critical reading of this thesis by Jessica and Maria, thank you very much girls!

Many thanks go, as well, to Andreas Hauser for his help in integrating his tool in this project.

I would, also, like to thank all the testers of the presented program for providing helpful comments and suggestions and all the members of the Söding, Tresch and Gagneur groups for creating such a great atmosphere and just having a good time together.

I highly appreciate the financial and personal support of the Deutsche Telekom Stiftung, which made this work possible.

Last but not least I would like to thank my son for giving me so much love, that supported me a lot through all these years. I would also like to thank my father for his help and suggestions and all my friends and family for the support, trust and encouragement.

Summary

Fueled by rapid progress in high-throughput sequencing, the size of public protein sequence databases doubles every two years. The widely used collection of protein sequences UniProt Knowledge Base (UniProtKB) contains 80 M sequences by now and has an even faster growth lately. It can be expected that this trend will continue since the next generation sequencing methods are becoming cheaper and faster every year.

The ever larger and more redundant databases lead to longer search times and bloated results lists that become tedious to analyse manually. A more subtle but important problem is that sequence space is sampled very unevenly by sequences in the database, which leads to biased results and at times limited sensitivity of iterative profile sequence searches.

Clustering sequence databases can help to organize sequences into homologous (i.e. evolutionarily related) and functionally similar groups. Searching through clustered databases can therefore improve the speed, sensitivity, and readability of sequence similarity searches. Clustered databases also provide a basis for very sensitive HMM-HMM search with HMMer or HHblits.

To cluster a set of sequences, all pairwise similarities are required, leading to quadratic time complexity with respect to the number of sequences. To cluster UniProtKB with 80 M sequences, 6.4×10^{15} pairwise comparisons are needed! In addition, the Smith-Waterman algorithm, which is the standard tool to compute sequence alignments, takes a time that scales linearly with the product of both sequence lengths. To speed up searches, many tools contain very fast alignment-free prefiltering heuristics. Even these fast methods are impracticably slow for databases beyond a million sequences even on large computing clusters. For instance, NCBI BLAST would need about 58 years for the clustering of a recent UniProtKB version (54 M sequences) on a computer with 32 cores. Therefore, fast, sensitive, and accurate clustering algorithms are urgently needed.

Several clustering methods have been developed in recent years. They all contain a fast prefiltering heuristic that rejects most of the non-homologous sequences before comparing the remaining sequences using a slow Smith-Waterman alignment algorithm. The prefilter of all fast clustering methods is based on the assumption that similar sequences share several short subsequence (k-mer) matches.

The most widely used clustering method is CD-HIT, developed in 2002. CD-HIT is able to cluster a protein database down to 50% sequence identity (i.e. 50% of aligned residues are identical). However, with lower sequence identity its prefitlering step becomes increasingly inefficient and the program gets slow. Even at low sensitivity settings, CD-HIT is too slow

for large databases containing tens of millions of sequences, needing weeks of computation time.

Another sequence clustering method developed recently is USEARCH. Usearch is very fast and needs only days to cluster the UniProtKB down to 50% pairwise sequence identity. However, it is not parallelized and therefore can not make use of the growing number of compute cores in modern CPUs. Moreover, USEARCH was not designed to detect sequences at similarities lower than 50% sequence identity.

Here I present MMseqs, a very fast and sensitive clustering and sequence search method. It implements a prefiltering that sums up the scores of similar k-mer matches between two sequences. The second step after the prefiltering is a fast implementation of the Smith-Waterman alignment algorithm. We use single-instruction-multiple-data (SIMD) instructions that parallelizes computations on arithmetic vector processors contained in all modern CPUs. As the final step, a clustering is calculated based on the sequence similarities resulting from the alignment step.

In addition, MMseqs offers a cascaded clustering workflow that is faster and more sensitive than our simple, single-step clustering. This workflow clusters a database incrementally, in three steps, starting with a very fast but low-sensitivity clustering step and ending with a slower but high-sensitivity step. It is therefore both faster and more sensitive than the simple clustering.

Finally, MMseqs offers the possibility to update the database clustering by adding new and removing deleted sequences without the need to recalculate all pairwise similarities and keeping the cluster identifiers stable. Clustering the UniProtKB from scratch takes hundreds of CPU hours and is impractical to keep up even using MMseqs. But since hundreds of thousands of new sequences are added to the UniProtKB database each week, frequent updating of the clustered database is important. The MMseqs updating workflow can compute an updated clustering of UniProtKB within hours instead of many days.

MMseqs is able to cluster a database down to 20-30% sequence identity, two to three orders of magnitude faster than BLAST-based clustering. Clustering the UniProtKB with 54 M sequences took 8 days and produced 6.3M clusters. In our clustering benchmark on a dataset containing sequences with 30-80% sequence identity, MMseqs at a high-sensitivity setting produced only 2% more clusters than BLAST while being 96 times faster. At lower sensitivity setting, MMseqs produced twice as many clusters as BLAST while being 140 times faster.

MMseqs can also be used as a stand-alone batch sequence search tool for searching sequences down to 20-30% sequence similarity in large sequence databases. In our tests, MMseqs had the best speed-to-sensitivity tradeoff, being orders of magnitude faster than the more sensitive tools SWIPE (SIMD-accelerated Smith-Waterman) and BLAST, while maintaining comparable sensitivity. On short sequence such as those obtained from next-generation sequencing, the MMseqs' sensitivity approached that of BLAST.

Contents

List of Figures

List of Tables

1. Introduction to homology searching and clustering

Proteins are biological molecules with a vast amount of functions. Proteins participate in virtually every process within the cell. Many proteins are enzymes and catalyze biochemical reactions. Others have structural or mechanical functions, for example in maintaining cell shape. Other proteins are important in cell signaling, immune responses or the cell cycle.

Proteins are polypeptides, i. e. each protein consists of a chain of amino acids. The sequence of amino acid residues in a protein is defined by the sequence of a gene, which is encoded in the genetic code. In general, the genetic code specifies 20 standard amino acids, although there are some non standard amino acids. The amino acid chain folds in space into a three dimensional structure which determines the function of the protein.

There is a very large amount of possible protein sequences. A protein of an observed average length 350 can have 20^{350} possible sequences. However, only a very small fraction of them can fold into a stable structure (Söding and Lupas, 2003). Therefore, the emergence of a protein with a certain function *de novo* is very unlikely. Proteins evolve, emerging from ancestor proteins through deletions, insertions and mutations in the protein sequence. Two proteins with a common ancestor are said to be homologous. Homologous proteins in most cases have very similar three dimensional structure and function due to the common ancestry, even when their sequence diverged considerably. Information from homologous proteins is used to assign functional annotation to proteins with unknown function, to determine their three dimensional structure and to study mechanisms of evolution (Loewenstein et al., 2009). Protein regions that are highly conserved are likely to be crucial for its function, like catalytic sites.

Homology detection is a basic procedure in computational biology. Using homology inference, it is possible to predict e. g. metabolic pathways, secondary structure (Rost and Sander, 1993), protein folding into 3-dimensional structure (Söding et al., 2005) or effects of single point mutations (Bromberg and Rost, 2007). The easiest way to identify homologs is pairwise comparison of proteins for finding the shortest sequence of mutation, deletion and insertion events that separate one protein sequence from another. The annotation of two protein sequences with these events and assigning a similarity score is called a pairwise sequence alignment. Ideally, the biologically most likely alignment gets the highest score. For the detection of the optimal alignment of two protein sequences, one needs a scoring scheme, describing the cost of converting one amino acid into another and inserting gaps in

the alignment which describe insertion/deletion of an amino acid.

1.1. Scoring models, gap penalties

A pairwise alignment of sequences should detect if the two sequences have a common evolutionary ancestor from which these sequence diverged by a series of mutations, deletions and insertions of amino acids. Some amino acids share similar biochemical properties and are more likely to mutate into one another, like charged amino acids. Other mutations occur very rarely in the nature, like a mutation of a charged into a hydrophobic amino acid. The alignment score of two sequences should be a good indication if the sequences are stemming from a common ancestor, i. e. if the sequence of the matches, mismatches and gaps in the alignment is likely to occur due to common ancestry or only occurs by chance.

Common scoring schemes for pairwise sequence comparison assume that all mutations occur independently from each other at each position in the sequence. The alignment score of two sequences is the sum of the scores for the matches/mismatches at each position and penalties for gaps. Furthermore, conservative substitutions that are observed often should get a positive score and non-conservative substitutions that occur rarely in real world alignments should be penalized with a negative score. The scores are recorded in a 20×20 amino acid substitution matrix which is symmetric, i. e. it is assumed that the probability of an amino acid a to mutate into an amino acid b is equal to the probability of b to mutate into a.

A substitution score compares the probability of a and b to be related to the probability of a and b to be aligned only by chance. The former is often related to as the match model M, and the latter as the random or null model R. In the random model R the alignment of a and b is a product of their background probabilities: $P(a,b|R) = f(a)\, f(b)$. In the match model M, the alignment of the two residues a and b occurs with a joint probability $P(a,b|M) = p(a,b)$. The alignment score for a and b is a *log-odds-ratio* :

$$\begin{aligned} s(a,b) &= \log\left(\frac{P(a,b|M)}{P(a,b|R)}\right) \\ &= \log\left(\frac{p(a,b)}{f(a)\, f(b)}\right) \end{aligned}$$

To be able to calculate $s(a,b)$, we need the background probability $f(a)$ for each amino acid a and the joint probability $p(a,b)$ for each amino acid pair (a,b) to be aligned. $f(a)$ is easy to calculate from unbiased protein sequence data. However, $p(a,b)$ depends on the evolutionary distance of the sequences. The evolutionary distance of two sequences is commonly measured as the sequence identity, i. e. the percentage of identical residues in the pairwise sequence alignment.

PAM was one of the first amino acid substitution matrices (Dayhoff et al., 1978). The

authors first measured the substitution probability $p(a,b)$ for very similar sequences with 1% accepted mutations (PAM1). Then, they extrapolated the substitution matrix to more diverse sequences to derive matrices up to PAM250. However, the extrapolation does not work well for diverged sequences with low sequence identities.

This problem was addressed by the BLOSUM matrices (Henikoff and Henikoff, 1992). The authors generated the BLOCKS database consisting of homologous sequences aligned without gaps and measured the joint probability $p(a,b)$ for different maximum sequence identities. BLOSUM62 matrix, i. e. a matrix derived from alignments with maximum 62% sequence identity, is now a default substitution matrix in many applications.

For the gap penalties, there are two widely used scoring schemes. In the first scheme, a linear gap penalty that assigns the same negative score $-d$ to the gap at each position in the sequence:

$$\gamma(g) \quad = \quad -gd$$

The second scheme involves a more sophisticated affine gap penalty that assigns a higher penalty $-d$ to opening a gap and a lower penalty $-e$ to the gap extension:

$$\gamma(g) \quad = \quad -d - e\,(g-1)$$

Affine gap penalties take into account the observation that in real alignments one long gap is more likely to appear than many short gaps.

1.2. Pairwise sequence alignment

1.2.1. Global alignment

Given a scoring scheme, the challenge is to find an optimal alignment between the two sequences, i. e. the alignment with the highest score. However, for two sequences x and y with lengths n and m there are $\binom{n+m}{n}$ possibilities for a non-redundant alignment. Therefore, the number of possible alignments grows exponentially with sequence length, so it is computationally not feasible to search for the best alignment by just generating all possible alignments and identifying the one with the best score. However, the vast majority of these alignments are not meaningful and have very low scores.

The solution is dynamic programming, i. e. to extend a subalignment of x and y from shorter subalignments. This algorithm was developed by Needleman & Wunsch in 1970 (Needleman and Wunsch, 1970). A subalignment $x_1,\ldots,x_i$ and $y_1,\ldots,y_j$ can arise in three ways: (1) by extending the alignment of x_i, y_{i-1} with a gap in x, (2) by extending the alignment of x_{i-1}, y_i with a gap in y and (3) by extending the alignment of x_{i-1} and $y_{i-1}$with

a match x_i, y_i. From this follows a recursive formula for the calculation of the dynamic programming matrix H, where the cell $H(i,j)$ records the best score for the subalignment $x_1,\ldots,x_i$ and $y_1,\ldots,y_j$

$$H(i,j) = \max \begin{cases} H(i-1,j-1) + s(i,j) \\ H(i-1,j) - d \\ H(i,j-1) - d \end{cases} \tag{1.1}$$

Starting with an empty alignment, the algorithm fills the matrix H from the upper left corner to the lower right corner. After filling the whole matrix, the score of the best *global alignment*, i. e. where each position in x is aligned either with a position in y or with a gap, is recorded in the matrix cell $H(n,m)$.

The actual alignment of x and y, i. e. the sequence of matches in gaps in x and y, is built by performing a traceback on the dynamic programming matrix H. It is built in reverse by starting at cell $H(n,m)$ and identifying in each step which of the three possible cells $H(i-1,j-1)$, $H(i-1,j)$ and $H(i,j-1)$ has contributed to the value of $H(i,j)$.

1.2.2. Local alignment

Real biological sequences are often only similar in one part of their sequence, for example due to a shared homologous domain, but unrelated otherwise. In this case, we want to find a best *local alignment*, ignoring the rest of the sequences. The algorithm for identifying the best local alignment was developed by Smith & Waterman in 1981 (year of my birth) (Smith and Waterman, 1981). It allows $S(i,j)$ to become 0 if all other options lead to the score dropping below 0:

$$H(i,j) = \max \begin{cases} H(i-1,j-1) + s(i,j) \\ H(i-1,j) - d \\ H(i,j-1) - d \\ 0 \end{cases} \tag{1.2}$$

After the filling of S the cell $S(k,l)$ with the best local alignment score is identified. To construct the full local alignment, a traceback is performed analogously to the global alignment traceback with the difference that it stops when the score reaches the value 0.

1.2.3. Local alignment with affine gap penalties

Until now, we used linear gap penalty to compute the best alignment for x and y. For using affine gap penalties, Gotoh developed

$$H(i,j) = \max \begin{cases} E(i,j) \\ F(i,j) \\ H(i-1,j-1) + s(i,j) \\ 0 \end{cases} \tag{1.3}$$

where the first two terms describe closing the gap in x, and y, respectively, and the third term describes elongating the alignment with a match between x_i and y_j. The affine gap matrices are filled as follows:

$$E(i,j) = \max \begin{cases} E(i,j-1) - e \\ H(i,j-1) - d \end{cases} \tag{1.4}$$

$$F(i,j) = \max \begin{cases} F(i-1,j) - e \\ H(i-1,j) - d \end{cases} \tag{1.5}$$

where the first term describes elongating a gap in x and y, and the second term describes opening a gap in x and y, respectively.

All presented dynamic programming algorithms have time and space complexity $O(n\,m)$. This makes them unfeasible for the use with large sequence databases often containing millions of sequences.

1.3. Alignment acceleration using SIMD parallelization

Smith-Waterman alignments (Section 1.2.3) find the best possible alignment between two sequences. However, the computation of pairwise local sequence alignments is slow due to quadratic run time complexity. Therefore, usually it is not feasible to calculate Smith-Waterman alignments for a large amount of query and database sequences (in the order of hundreds of thousands and more). During the last 20 years, several efforts were made to achieve a speed up using SIMD parallelization.

Several efforts were made to accelerate the calculation of Smith-Waterman alignments with Gotoh improvements using Single-Instruction-Multiple-Data (SIMD) processors. The most notable are the implementation of Farrar (Farrar, 2007) which uses intra-sequence parallelization, and Rognes (Rognes, 2011) using inter-sequence parallelization.

1.3.1. SIMD technologies

A SIMD (single instruction multiple data) instruction is able to perform a simple arithmetical operation on a vector instead of a single value in one processor cycle. These arithmetical operations include basic integer and floating point operations such as addition and multiplication, number comparison operations, data shuffling, data-type conversion and bit-wise logical operations.

Most modern CPU designs include SIMD instructions. Well known examples are the SSE (Streaming SIMD Extensions) instruction set of Intel, introduced in 1999, and its extensions SSE2, SSE3, SSSE3 and SSE4, introduced in 2001, 2004, 2006 and 2007, respectively. All SSE instruction set iterations are also supported by modern AMD processors. SSE work with 128-bit registers (XMM registers), where different data types can be stored. For example, an XMM register can hold four 32-bit single-precision floating point numbers. Then, a single instruction is performed in parallel on all numbers stored in an XMM register.

SSE2 expand the usage of the XMM registers to be more flexible. XMM is then able to hold two 64-bit double-precision floating point numbers, or two 64-bit integers, or four 32-bit integers, or eight 16-bit short integers, or sixteen 8-bit bytes or characters.

A recent instruction set is AVX (Advanced Vector Extensions), proposed by Intel in 2008 and also supported by newer AMD processors. The width of the SIMD register file in AVX is increased from 128 bits to 256 bits. Therefore, AVX is able to process twice as much data in one CPU cycle as SSE instruction sets.

1.3.2. Smith Waterman SIMD implementations

One of the first vectorized Smith-Waterman implementations is the implementation of Wozniak in 1997 (Wozniak, 1997) running on the Sun Ultra SPARC using its SIMD instructions. It subdivided the dynamic programming matrix in vectors held in 128 bit registers lying on anti-diagonals of dynamic programming matrix (Figure 1.1). A 2-fold speedup was achieved over the original Smith-Waterman implementation. This implementation has no conditional branches in the inner loop, therefore the runtime does not depend on the scoring matrix or the gap penalties.

The other notable Smith-Waterman implementation was presented by Rognes and Seeberg in 2000 (Rognes and Seeberg, 2000). It used MMX SIMD instructions to parallelize the calculation of the dynamic programming matrix. The vectors holding the dynamic programming matrix values were placed simply along the matrix (Figure 1.1). The main speed up was achieved by generating a query profile once for the entire database search. A drawback are conditional branches in the inner loop slowing down the program. This implementation achieved a 6-fold speedup over optimized non-SIMD implementation.

However, both implementations have a major drawback: for the calculation of the dynamic programming matrix values, single values have to be extracted from SIMD registers

and combined (Figure 1.1). This slows down the program, since the extraction of values from vectors in SIMD calculations instead of operations on SIMD registers as a whole is expensive.

Striped Smith-Waterman

This implementation of the Smith-Waterman algorithm with the Gotoh improvements for affine gap penalties developed by Michael Farrar (Farrar, 2007) uses SSE2 instruction set for the parallelization of the dynamic programming matrix calculation. Striped Smith-Waterman implementation achieves a speedup of 2-8 times over the Wozniak and Rognes&Seeberger SIMD implementations. The main algorithmic improvements are the usage of the query profile instead of many substitution matrix lookups, removing of the most single-value dependencies between the SIMD registers during the matrix calculation and removing of the conditional branches in the inner loop.

Before the calculation of alignments of the query with database sequences starts, a scoring profile for the query is generated, i. e. at each query position the row of the substitution matrix for the amino acid at that position is stored. Therefore, the calculation of the next $H_{i,j}$ (eq. 1.3) requires just an addition of the pre-calculated score to the previous $H_{i.j}$.

Striped Smith-Waterman uses two different data types during the calculation. Due to the limitation in the SSE2 instruction set, unsigned 8-bit integers are used for the calculation of the values in the dynamic programming matrices. To make use of the whole unsigned 8-bit integer range, the query profile is biased to the smallest value in the scoring matrix. The unsigned 8-bit integer range is sufficient for most Smith-Waterman alignment score

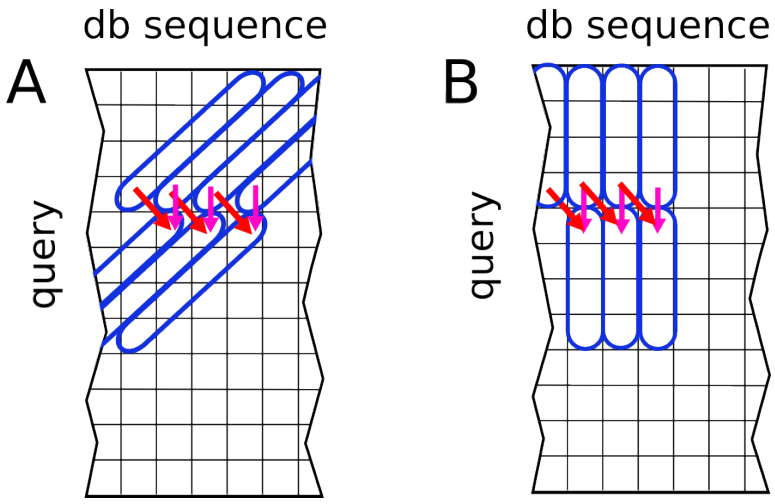


Figure 1.1.: **SIMD implementations of Smith-Waterman alignment: (A)** Wozniak - vectors along the anti-diagonal, **(B)** Rognes and Seeberger - vectors along the query. The values held in a single SIMD register are circled. For simplicity it is assumed that one vectory contains only 4 values. Updates requiring extraction of single values are shown with arrows (red: reads of matrix H, magenta: reads of matrix F).

calculations. After the calculation of the maximum score, the bias is subtracted from the score again. When the score is greater than the maximum possible 8-bit integer value, the calculation is repeated again with 16-bit integers. However, the calculation is then only half as fast, since a SIMD register can store and process half as many 16-bit integers in every step, compared to 8-bit integer values.

The algorithm uses a striped layout of the query for avoiding computationally expensive shifting and extracting values held in SIMD registers. The query is divided into segments, the number of segments is the number of entries in the SIMD register, i. e. 16 segments when 8-bit integers are used, and 8 segments when 16-bit integers are used. Then, dynamic programming matrix values are stored in matrix such that one SIMD register contains one value per segment (Figure 1.2). Due to this layout, the next column of the matrix can be updated by adding the profile query scores vector-wise without shifting or extracting values. The only exception per database position where a shifting of values is necessary is the last vector.

The gap extension matrix E can be updated simply from left to right. It uses the same segmentation as in the H matrix. For the updating, it uses the previously calculated columns of H and E and adds the gap penalties to calculate the current column.

A column of the gap extension matrix F is calculated from the top to the bottom. The conditional branches in the inner loop should be avoided, since the penalty for the wrong branch prediction, used by the modern processors, is especially high in the inner loop. Therefore, the algorithm uses lazy evaluation of the gap extension matrix F. F only starts to influence the value of H when H becomes greater than $g + e$, what is mostly not the case. After the calculation of H without considering F, the values of F are checked to see if they are high enough to influence H and a recalculation is necessary. If the scores in F are high enough, F and H are recalculated, since F can then change the values of H.

Smith-Waterman with inter-sequence parallelization

Here the algorithm developed by Rognes (Rognes, 2011) parallelized the computation of the Smith-Waterman alignments with Gotoh improvements over multiple database sequences instead of parallelizing the computation of a single alignment like previously described approaches (Figure 1.2).

First, the scoring profile of the query sequence is computed similar to the algorithm of Farrar to remove the scoring matrix lookups. Then, the alignments of a query sequence to multiple database sequences are computed. Residues from 16 different database sequences are processed in parallel (Figure 1.2 shows it on an example of processing 4 database sequences in parallel). The values of the three dynamic programming matrices H, E and F are calculated independently for each database sequence in parallel. In each step, dynamic programming matrices are filled for 4 database residues, before going to the next query sequence residue. The reason is the reduction of the running time through more efficient

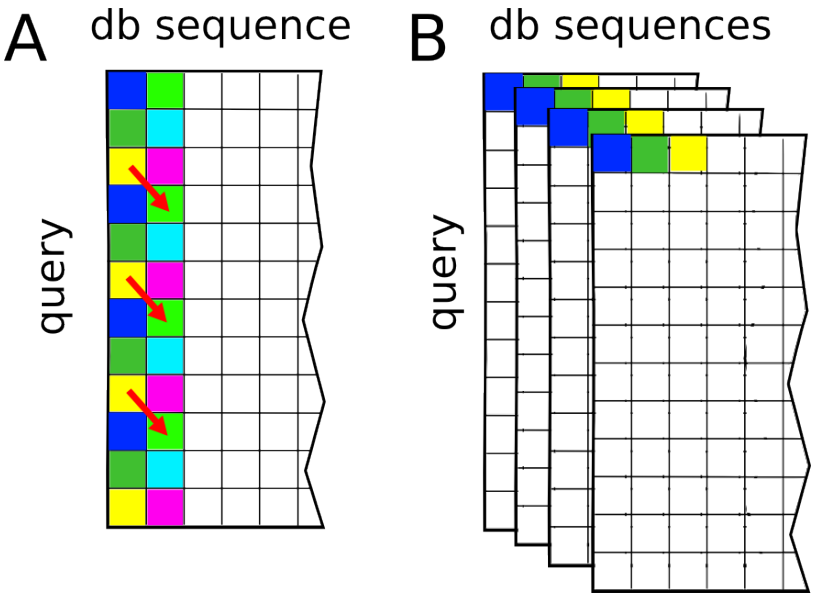


Figure 1.2.: **SIMD implementations of Smith-Waterman alignment.** For simplicity it is assumed that a vector contains only 4 values. The values of the matrix belonging to one vector are marked in the same colour. **(A)** Farrar intra-sequence parallelization. The query is processed not linearly, but by partitioning the query into 4 parts. One vector is containing 4 values from each part of the query. Due to the arrangment of the matrix values in the vectors, the updating of the H matrix along the database sequence can be done mostly without extracting single values. Only for one vector per database sequence position (yellow) the values have to be shifted (red arrows) for the updating the matrix at the next database sequence position (light green). **(B)** Rognes inter-sequence parallelization. Each vector holds scores for residues from 4 different database sequences (for simplicity, in practice vectors of 16 elements are used). The 4 database sequences are aligned to the query simultaneously.

cache usage when processing multiple database residues consecutively.

After having processed the 4 residues block for the whole query sequence, the algorithm checks if one of the database sequences ended. In this case, the score of the database sequence is recorded and the next database sequence is loaded in the channel. For sequences which length is not a multiple of 4, the sequence is padded with null symbols. Then, the check for the sequence end does not need to be carried out during the processing of the 4 residues block.

For the speed, the most inner loop, i. e. the updating of the three dynamic programming matrices H, E and F, is written in assembly instructions.

The implementation of the algorithm (SWIPE) is the fastest among the SIMD Smith-Waterman implementations. It achieves running times 2-18 times faster than the Farrar implementation, depending on the input data. However, the main limitation of SWIPE is the necessity of a minimum amount of the database sequences per query in order to reach its maximum speed.

1.4. Sequence profiles and HMMs

In the scoring scheme described above, we assumed that all the positions in the sequence evolve independently and have an equal probability for mutations. However, this assumption does not hold under the real life conditions. Some positions in the protein are more likely to mutate, like unstructured loops that connect functional units of the proteins. Others are strongly conserved, like an active site of the enzyme. To take the different mutation probabilities of the different sequence positions into account, sequence profiles are used for the similarity scoring.

A sequence profile is generated from a multiple sequence alignment of the query sequence with its homologs. For each alignment position, different mutation probabilities are derived from the frequency of the different amino acids at this position. The resulting sequence profile can then be aligned with some target sequence. During the alignment, the match scores $s(i,j)$ are not taken from an amino acid substitution matrix like BLOSUM62, but from the sequence profile at position i. Therefore, the resulting alignment considers the conservation of the residues stated in the sequence profile.

An extension of this approach is the use of Hidden Markov Models (HMMs). HMMs not only determine position specific mutation probabilities, but also position specific insertion and deletion probabilities for the query MSA. Therefore, an alignment to a target sequence becomes even more accurate. HMM-HMM alignment has proven to be most powerful method for remote homology detection in proteins (Eddy, 1996). The fastest available HMM-HMM alignment tool is HHsearch (Söding, 2005).

There exist faster HMM-HMM search methods that use fast prefilters to reduce the size of the search space before the actual HMM-HMM search is run. HMMER (Sonnhammer et al., 1998), developed by Sean Eddy in 1998, uses fast prefilters before calculating HMM-sequence alignments to accelerate the search. HMMER3 (Eddy, 2011) adds the iterative HMM-HMM search and improves the performance by using vector programing. HMMER3 is approximately three times slower than PSI BLAST.

HHblits (Remmert et al., 2012) is an iterative HMM-HMM search tool that uses a clustered database to generate an HMM database and accelerates the search by using several fast prefiltering steps before computing a most sensitive and costly HMM-HMM alignment for the rest of the database.

1.5. Fast sequence search heuristics

In the last sections, I presented exact sequence comparison with the Smith-Waterman algorithm. However, the calculation of exact pairwise alignments is too slow for many real life applications where often millions of sequences have to be compared. Therefore, over the last decades, many fast heuristics for the sequence search were developed.

1.5.1. FASTA

FASTA is a fast heuristic for sequence comparison that was developed by Pearson and Lipman in 1988 (Pearson and Lipman, 1988). In the first step, it counts short identical words between the query sequence and database sequences and identifies the best diagonals with the most proximate matches which indicate regions of high similarity. In the second step, it rescores these regions using a scoring matrix and saves the best scoring initial regions. In the third step, FASTA checks if several initial regions can be joined together using their score and gap penalty for the distance between the initial regions. Finally, a local alignment around the best match is recalculated and reported.

1.5.2. BLAST and PSI-BLAST

BLAST is a fast search tool with a high sensitivity which searches for homologous sequences of a query sequence within a sequence database (Altschul et al., 1990). BLAST search algorithm is based on the observation that a sequence pair with a high-scoring alignment has multiple short words, so-called k-mers, in common, which lie on the alignment path within a short distance.

First, all 3-mers of the query sequence are indexed. For each 3-mer, similar 3-mers are generated based on their similarity score until a certain similarity threshold. The resulting 3-mer list is organized into a search tree to make the search more efficient. Then, the database is searched for the high-scoring 3-mers and each match is recorded.

Then, BLAST tries to extend 3-mer matches into a so-called high-scoring segment pair

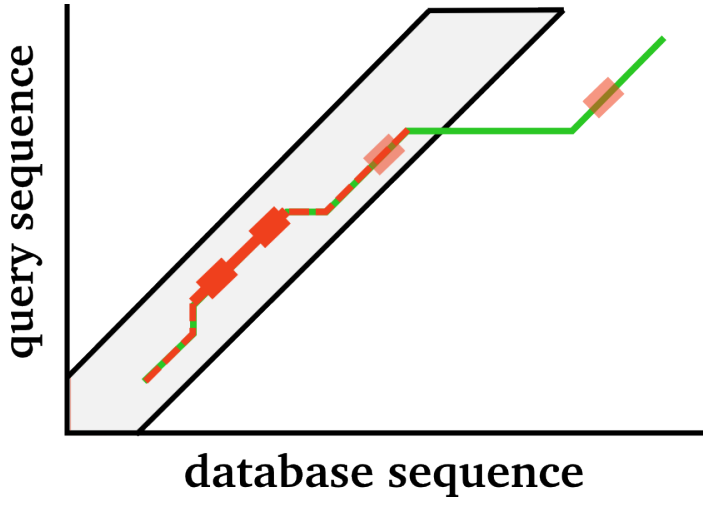


Figure 1.3.: **BLAST algorithm.** The green line is the optimal alignment between the query and the database sequence. BLAST searches for the matches of similar 3-mers between the sequences (red rectangles) and finds two 3-mer matches within a short distance on the same diagonal. The two exact matches are extended with an ungapped alignment to form a HSP (red line). The HSP is then extended with a bounded Smith-Waterman alignment restricted to the grey area around the HSP.

(HSP). The original version of BLAST extended each 3-mer match between the query and the database sequence until the score dropped below a certain threshold. However, this approach involves the dilemma between speed and sensitivity: short k-mers are required to be sensitive, but at the same time short k-mers produce a large amount of chance matches that need to be verified. Therefore, the newer version requires two non-overlapping k-mer matches on a diagonal within a window of 40 residues which reduces the number of chance matches. The matches are then extended to form a HSP. The extension procedure is based on the Smith-Waterman alignment, but is restricted to the neighborhood of the found HSPs. All resulting alignments with a score above a threshold and a high statistical significance are then reported. The whole algorithm is illustrated in Figure 1.3.

PSI-BLAST (Altschul et al., 1997) is an extended version of BLAST that first runs a standard BLAST algorithm. Then, it builds sequence profiles from each query sequence and similar sequences found in the first step. Then, it searches the database again using the generated sequence profiles as queries. The accumulation of sequences, generation of profiles and search can be repeated iteratively, each step resulting in a more sensitive search.

Karlin-Altschul statistics

Karlin-Altschul statistics (Altschul and Gish, 1996) provide a theory for computing the probability that a local alignment with a certain score will be found between two sequences of a database at random.

It is assumed that matches of amino acids between two sequences occur independently. In addition, the expected score of a match should be negative, since otherwise local alignments with a considerably high score could occur by chance.

It is known that the sum of a large number of independent identically distributed random variables tends towards a normal distribution, and the maximum of a large number of independent identically distributed random variables tends towards an extreme value distribution. Under the assumption that local alignment scores are independent identically distributed, the optimal local sequence alignment scores should obey an extreme value distribution. For a query sequence of length m and a database sequence of length n obtaining a local ungapped alignment score S, the expected number of ungapped alignments E (e-value) obtaining the score S or better is given by the formula

$$E = K m n e^{-\lambda S} \tag{1.6}$$

where K and λ are the factors describing the scale of the search space size and the scoring scheme, respectively. Therefore, to use the formula 1.6 for the e-value computation, we need to know K and λ, i. e. to normalize the raw scores to obtain so-called “bit scores”:

$$S' = \frac{\lambda S - \ln K}{\ln 2}$$

Using bit scores instead of raw scores, the e-value can be calculated with the formula

$$E = m\,n\,2^{-S'} \tag{1.7}$$

To calculate the e-value of a local alignment of a query with a sequence contained in a database of N sequences, there are two different approaches. In the first approach, it is assumed that every sequence of the database is equally likely to be related to the query. In this case, to calculate an e-value for an alignment found during a database search, one has to multiply the e-value in the formula 1.9 with the number of database sequences:

$$E = N\,m\,n\,2^{-S'} \tag{1.8}$$

The other approach assumes that long sequences are more likely to be related to the query than short sequences (e. g. because longer proteins often contain multiple distinct domains). In this case, the database is treated as a very long sequence of length L, where L is the sum of lengths of the database sequences, the e-value for an alignment found during a database search is

$$E = m\,L\,2^{-S'} \tag{1.9}$$

BLAST uses the second approach for computing e-values.

Computational tests and analytical results suggest that the same e-value calculation applies also to gapped local alignments.

1.5.3. BLAST-like searches

There are several methods implementing search algorithms very similar to BLAST. They are usually either faster than BLAST, but much less sensitive, or they achieve only a very modest speed-up.

PLAST

PLAST (Nguyen and Lavenier, 2009) is a faster alternative to the BLAST algorithm, achieving a speedup ranging from 3 to 6 with a similar level of accuracy. PLAST indexes both query and target databases and uses the SIMD parallelization and multi-core parallelization

to achieve the speed-up. The seed searching and extension algorithm differs from BLAST only slightly. PLAST shows the best performance when searching large databases.

BLAT

Another BLAST-like search tool is BLAT (Kent, 2002) which indexes non-overlapping k-mers and then uses a BLAST-like extension of the matches, eventually aligning the regions likely to be homologous. BLAT achieves a 50 fold speedup compared to BLAST, at the cost of the lower search sensitivity.

PSimScan

PSimScan (Kaznadzey et al., 2013) is a search tool building a k-mer index of the database and then extending these matches in several steps in a BLAST-like manner. Finally, alignments for the remaining sequence pairs are computed. Its speedup compared to BLAST ranges from 5 to 100 depending on the chosen parameters.

RAPSearch

RAPSearch (Ye et al., 2011; Zhao et al., 2012) uses a reduced amino acid alphabet and a collision-free hash table to identify common substrings of flexible length which eventually trigger a seed extension with BLAST extension algorithm. RAPSearch achieves ~20-90 times speedup as compared to BLAST, but its sensitivity is considerably lower.

1.5.4. UBLAST

UBLAST (Edgar, 2010) is a very fast sequence search algorithm similar to BLAST. It searches for short word matches between sequence pairs and extends the found matches by computing a local alignment around them. It is designed to find sequence pairs with low sequence similarity below 50%.

UBLAST uses matches of short words of length 3 (3-mers) to find candidate sequence pairs. To increase sensitivity, it uses spaced seeds - k-mers with gaps which are designed to reduce the correlation between positions of the k-mer. Another feature for increasing the sensitivity is the usage of reduced amino acid alphabet for the k-mers, which allows not only exact matches but also matches between similar amino acids.

The database is indexed to enable fast access to the 3-mers of the database sequences. The sequences of the database are ordered by the number of unique k-mers in common with the query sequence. Then, the matching 3-mer pairs are extended to HSPs using a BLAST-like algorithm. The similarity of a HSP is defined either as fractional sequence identity or e-value of the local alignment. For targets with similarity of the HSP above a given threshold, remaining regions are aligned using banded dynamic programming (Chao et al., 1992) and similarity is computed from the final alignment.

The search is terminated after a predefined number of targets were accepted (default 1) or rejected (default 8).

1.5.5. Ultra-fast search heuristics

Several sequence search heuristics were developed for very fast searches in large databases. Though being orders of magnitude faster than BLAST, they are only able to detect very close similarities, missing more distant homology.

Tachyon

Tachyon (Tan et al., 2012) search represents a sequence with a set of five contained 5-mers. Sequences sharing at least three 5-mers are then compared with a full-length sequence comparison method, which can be chosen by the user. Tachyon is about 200 times faster than BLAST, but produces about 15 times less hits when searching the NCBI NR database.

Simrank

Simrank (DeSantis et al., 2011) counts unique k-mers in the sequences. For each query, it returns the database sequences with the most similarities based on the k-mer counts, without calculating any alignment. Simrank is very fast, being 10 to 1000 times faster than BLAST depending on the dataset. However, it has a very low sensitivity and specifity compared to BLAST.

PAUDA

PAUDA (Huson and Xie, 2014) is a very fast method for the sequence search which builds an index from the reference database and encodes sequences with a reduced amino acid alphabet containing 4 letters. It achieves a 10 000 times speedup compared to BLASTX on a set of 246 M Illumina DNA reads. PAUDA succeeds in assigning only 33% of the number of reads assigned by BLAST.

1.6. Clustering

Clustering of the sequence data is often applied to large sequence databases as the first processing step.

Generally, clustering of a sequence database would require an all-against-all comparison of the sequences and then definition of groups (clusters) based on the similarity graph. However, for larger databases it is far too time-consuming to do an all-against-all comparison using Smith-Waterman alignments of the sequences.

The most straightforward way to cluster a database is to compute all-against-all sequence comparison with a sequence search method and to cluster the database with some clustering

algorithm based on the search scores. The most exact sequence search method is the computation of Smith-Waterman alignments (Smith and Waterman, 1981). However, the computation of all-against-all Smith-Waterman alignments and subsequent clustering is not feasible for larger databases, since it has quadratic time complexity in the number of database sequences and Smith-Waterman alignment has the quadratic time complexity in the sum of database sequence lengths.

1.6.1. BLAST-based clustering

There are multiple clustering algorithms based on the BLAST and PSI-BLAST scores and statistics (Enright et al., 2002; Heger et al., 2007; Nepusz et al., 2010).

Many clustering methods use FASTA or BLAST for this purpose. Some of these methods use the pairwise sequence similarities for hierarchical clustering (Krause et al., 2000; Miele et al., 2011; Rappoport et al., 2012; Yona et al., 2000). Others use them to cluster sequences into orthologous or functionally similar groups (Alexeyenko et al., 2006; Chen et al., 2010; Enright et al., 2002; Li et al., 2003; Powell et al., 2012; Remm et al., 2001; Tatusov et al., 2003). However, FASTA or BLAST would still need at least 70 years of even parallelized computation to cluster a sequence database like the present UniProtKB with ≈80 millions sequences from scratch.

SIMAP (Rattei et al., 2010) employs a 9-TeraFLOP distributed network of computers to regularly update their database of precomputed FASTA similarity scores for a large set of protein sequences. This database can be tapped to avoid the time-consuming sequence alignments (Powell et al., 2012), but it cannot be used for sequences not yet contained in SIMAP or when the clustering should depend on information other than sequence similarity scores.

Here, we present a few clustering methods able to cluster large databases in the order of millions of sequences down to low (30-50%) sequence identities from scratch. All the methods use a fast prefilter to sort out sequence pairs that are not similar at all. Based on the observation that homologous sequences share a certain number of short subsequences of length k (k-mers), not similar sequence pairs are sorted out and sequence pairs that passed the prefiltering are compared with a more sensitive pairwise sequence comparison method.

1.6.2. CD-HIT

CD-HIT is a widely used program for clustering biological sequences. It was first published in 2001 (Li et al., 2001) and refined in 2002 (Li et al., 2002b). CD-HIT uses a greedy incremental clustering algorithm. Each cluster is represented by its longest sequence, called the representative sequence. Before the clustering starts, the database is sorted by decreasing length. The first sequence is the representative sequence of the first cluster. Then each sequence is compared to the existing representative sequences. If a representative sequence

is similar enough to the query, the query is put in its cluster. Otherwise, the query forms a new cluster and becomes its representative sequence.

Sequence comparison step consists of a k-mer based prefiltering step and a subsequent Smith-Waterman alignment. In the prefiltering, number of k-mer matches between the query sequence and the sequences of the database is determined. k is chosen in the range $[2:5]$. For clustering protein sequences down to high sequence identities (more than 70%), 5-mers can be used. It makes the prefilter very fast and efficient, since a lot of sequence pairs can be excluded. For lower sequence identities, the k-mer size is reduced in order to make search more sensitive. However, the reduction of the k-mer size slows down the program because of the high probability of chance k-mer matches.

The sequence pairs that passed the prefiltering are aligned with banded Smith-Waterman alignment.

CD-HIT can be used for clustering protein and nucleotide sequences. In addition, it offers a version for comparing two different datasets (Li and Godzik, 2006). Recently, a new parallelized version of CD-HIT was implemented which uses all the cores of a computer (Fu et al., 2012).

1.6.3. USEARCH

USEARCH (Edgar, 2010) is a sequence search and sequence clustering algorithm similar to UBLAST, but designed to find high-identity hits, for proteins with sequence identity above 50%. It uses 5-mer matches to identify similar sequence pairs. USEARCH prioritizes the database sequences by the number of 5-mer matches, similarly to UBLAST, and then extends the matches with a local alignment.

The clustering algorithm is similar to CD-HIT. Initially, an empty clustering is created and the sequences of the database are ordered by length. The first sequence becomes the representative sequence of the first cluster. Each following sequence is compared to the existing representatives with USEARCH. If a match of sufficient similarity is found, the sequence becomes a member of its cluster. Otherwise it becomes the representative sequence of a new cluster.

1.6.4. SEED

SEED (Bao et al., 2011) is a very fast clustering tool based on the usage of block spaced seeds. It works only with Illumina reads. SEED can only handle short evolutionary distances, joining sequences into one cluster that differ by up to three mismatches an three overhanging residues from their virtual center. SEED is very fast, needing only about 4 hours to cluster 100 M short read sequences but it yields many more clusters as other methods such as CD-HIT and USEARCH.

1.6.5. kClust

kClust (Hauser et al., 2013) is the predecessor of MMseqs, the clustering method presented in the next chapter. It is able to cluster large protein sequence databases containing tens of millions of sequences. kClust uses a greedy incremental clustering algorithm as used in CD-HIT, sorting the database sequences by length and putting the next query sequence into the cluster of the most similar representative sequence.

kClust compares the query sequence to already found representative sequences in two steps. First, it reduces the number of considered representative sequences in the prefilter, by summing up the scores of the similar 6-mers instead of simply counting the number of exact k-mer matches as done in CD-HIT. The use of similar k-mers makes more sensitive searches possible, at the same time reducing the number of chance matches thanks to the used k-mer length. By adding the scores of similar 6-mers instead of simply counting them, the prefiltering can distinguish better between related and unrelated sequences.

For the sequences that pass the prefiltering step, pairwise alignments are calculated using a fast heuristic, k-mer dynamic programming (Mayer, 2007).

kClust has two major drawbacks. First, it is not parallelized - it uses only one core for the computations. Second, kClust does not offer the possibility to update the existing clusters with new sequences. This functionality would be very useful considering the growth of the major protein sequence databases, where hundreds of thousands of protein sequences can be added to a database within a week.

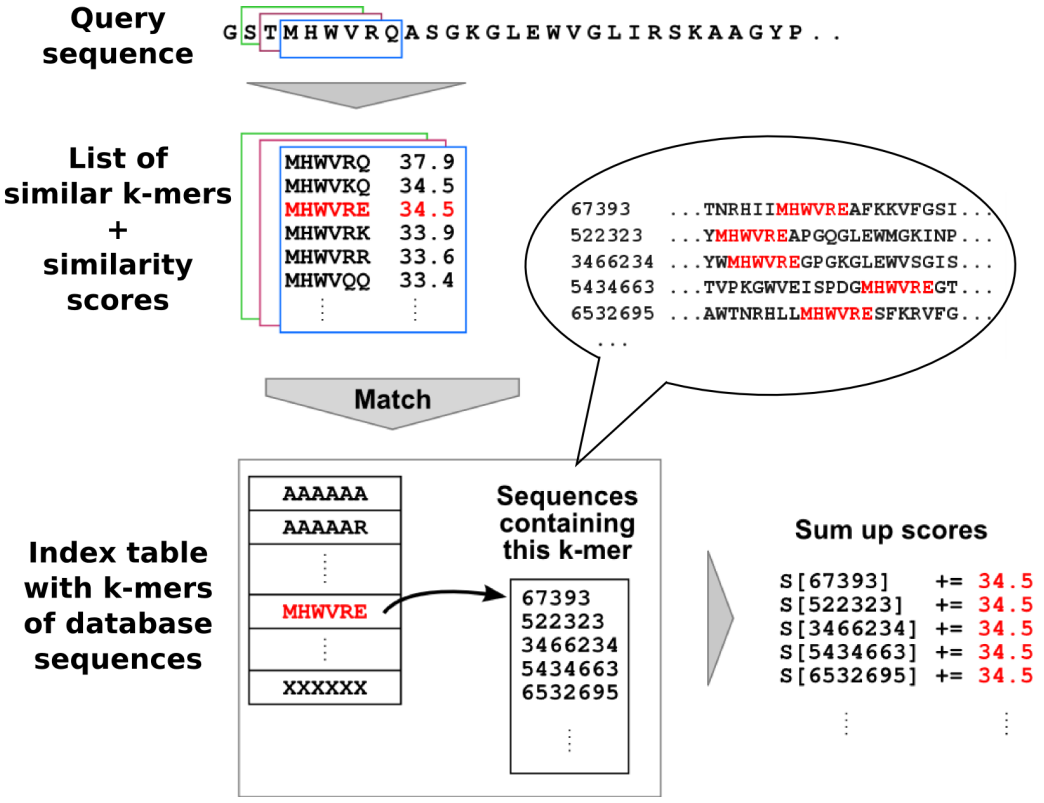


Figure 1.4.: **Overview of the kClust prefiltering algorithm.** The 6-mers of the database are stored in the index table prior to the calculation of the prefiltering scores. For each possible 6-mer, the index table contains a pointer to the list of database sequences containing this 6-mer. During the prefiltering scores calculation, the query sequence is processed from left to right. For the 6-mer at each position, a list of similar 6-mers is generated. Database sequences containing this 6-mer are identified using the index table and the similarity score of the 6-mers is added to the overall prefiltering score of this database sequence, stored in the array S. After the query sequence is processed, the best scoring database sequences passing the prefilter are extracted. (Figure taken from Hauser et al., 2013).

Part I.

Very fast and sensitive sequence search and clustering

2. Motivation

The rapid development of next-generation sequencing (NGS) strategies leads to a drastic increase of available sequence data at ever lower costs. New sequencing technologies from Roche/454, Illumina, Pacific Biosciences Inc., and Oxford Nanopore Technologies are able to produce data on the order of hundreds of giga base-pairs per machine day (Metzker, 2010). With the release of Illumina's HiSeq X Ten, for the first time a 1000$ genome has become reality (Hayden, 2014).

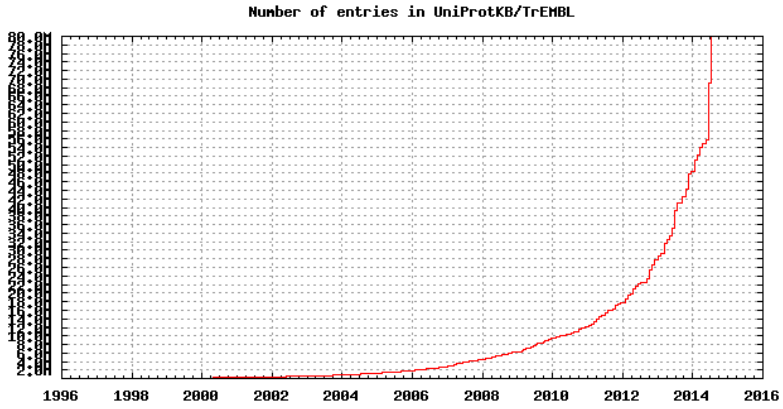
This development makes many applications possible which were limited by the low sequencing throughput in the past: metagenomic projects sequence whole bacterial communities, analyzing their taxonomic and genetic composition (Qin et al., 2010; Rusch et al., 2007; Yooseph et al., 2007). The 1000 Genomes Project (1000 Genomes Project Consortium et al., 2010) has sequenced over 1000 genomes and the sequencing of even larger datasets is planned in the future. This effort will provide a detailed image of human genetic variations, help us to understand common human deseases by analysing their association with genetic variation, and, based on the disease-associated biochemical and regulatory pathways, to develop personalized medical treatments (Koboldt et al., 2013).

The rapid growth of protein sequence data is reflected by the growth of the central repository of protein sequence data UniProtKB (Apweiler et al., 2004). UniProtKB contains about 80 M sequences as of end of July 2014, and has doubled every two years until recently, but the last year suggests even faster growth in the future (Figure 2.1a).

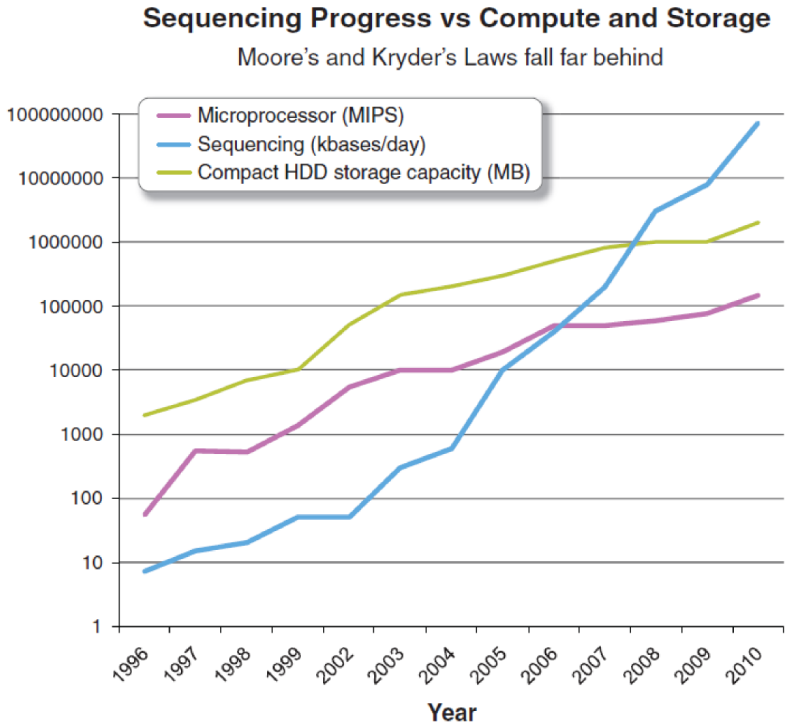
However, the flood of sequence data poses new challenges in their storage and analysis (Berger et al., 2013; Kahn, 2011; Kircher and Kelso, 2010). The sequence data accumulates more quickly than the growth in computing efficiency predicted by Moore's law (Figure 2.1b). Although the amount of protein sequence data grows rapidly, the rate of discovery of novel sequences declines. While one would expect the fast growth of the sequence databases and therefore the denser sampling of the sequence space to lead to better performance of sequence searches, the opposite seems to be true: The increase has led to stagnating or even negative returns, as measured by the ability to detect homologous sequences for structure or function predictions (Chubb et al., 2010).

Removing redundant sequences through clustering can partly alleviate this problem: Li et al., 2002a and Park et al., 2000 showed that sequence searches through clustered databases, which contain one representative sequence per cluster, improve the sensitivity of homology search methods. This is one reason for the popularity of the UniRef database, which provides clusterings of the UniProtKB at different sequence identity thresholds (Suzek

et al., 2007). Clustered databases also increase the sensitivity of profile-based sequence search methods (Li et al., 2002a).



(a) Growth of the UniProtKB database.



(b) The growth of the sequence data against the processor speed.

Figure 2.1.: **a)** Number of entries in the UniProtKB plotted over time (figure taken from http://www.ebi.ac.uk/UniProt/TrEMBLstats). **b)** A doubling of sequencing output every 9 months (blue) has outpaced and overtaken performance improvements within the disk storage (yellow) and high-performance computation (magenta) fields (figure taken from Kahn, 2011).

Furthermore, clustered databases, in which each cluster of homologous sequences is represented by a profile HMM, are a requirement for more sensitive sequence analysis methods based on the pairwise comparison of HMMs such as HHblits. HHblits is probably the most

sensitive protein sequence search method to date (Remmert et al., 2012). It calculates a profile HMM from the query sequence and searches for homologous matche to this query HMM in a database of profile HMMs computed from the MSAs obtained by clustering all UniProtKB sequences down to 20%-30% sequence identity. In subsequent iterations, sequences belonging to significantly similar database HMMs are added to the query multiple sequence alignment (MSA). Thanks to the additional information from homologous sequences in the MSAs, HMM-HMM comparison is more sensitive and accurate than profile-sequence comparison: Compared to PSI-BLAST (Altschul et al., 1997), HHblits is faster, up to twice as sensitive, and produces alignments with several percent higher accuracy.

For HHblits it is critical that only a very low number of clusters are corrupted by non-homologous sequences, since these can cause high-scoring false-positive matches. The clustering sensitivity is also important because MSAs with higher sequence diversity and higher information content are better at finding remotely related sequences. Also, the lower number of clusters results in shorter search times.

The established sequence search tools BLAST, PSI-BLAST (see section 1.5.2) and FASTA (section 1.5.1) are sensitive but are not fast enough to cluster the huge data amounts produced by NGS methods. Even faster methods were developed during the last years: A few faster search have been developed like BLAT, UBLAST and RAPsearch (see sections 1.5.3 and 1.5.4) but at the cost of a much lower sensitivity.

In addition, several ultra-fast sequence search methods exist (see section 1.5.5), but all of them lack sensitivity to do searches down to sequence identities below 70% (Li et al., 2012).

Most noteworthy for sequence clustering are the tools CD-HIT (see section 1.6.2), USEARCH (section 1.6.3) and kClust (section 1.6.5). Each of these methods has its own drawbacks: CD-HIT is fast if only very similar sequences are required to be clustered but becomes very inefficient below a sequence identity threshold of 70%. USEARCH and kClust are not parallelized and therefore not able to make use of the steady increase in the number of cores per CPU. Parallelization has become crucial in the last years, since the exponential growth of the CPU speed predicted by Moore's Law can now only be achieved by using the multiple cores of a computer in parallel (Sutter, 2005).

In the next chapters, we describe MMseqs, a very fast highly parallelized sequence clustering and sequence search tool. It implements a k-mer prefilter that sums up the similarity scores of similar k-mers, instead of counting exact k-mer matches. By analysing not only identical but similar k-mers, we obtain sufficient numbers of k-mer matches even at low sequence similarities. At the same time, the longer k-mers we use - between 5 and 7 - ensure very few chance matches and thus high search speeds.

MMseqs is around 1000 times faster than protein BLAST and sensitive enough to capture similarities down to less than 30% sequence identity. MMseqs owes its speed and sensitivity to the fast prefilter that sums up the scores of similar k-mers for sequence pairs. Then, a SIMD-parallelized Smith-Waterman alignment is computed for all sequence pairs that

passed the prefiltering. The alignment implementation is based upon the implementation of Farrar (Farrar, 2007). Prefiltering and alignment modules of MMseqs are the core of its functionality and can also be used for stand-alone sequence search. They are parallelized to use all cores of a computer to full capacity. A clustering is computed based on the sequence alignment results. In addition, MMseqs offers an updating function, which is very useful considering the fact that about five million sequences are added to the UniProtKB each month. Updating takes an already clustered version of the database and the new version of the sequence database. Then it adds new sequences to the clustering and removes deleted sequences without needing to calculate the whole clustering from scratch.

3. Description of algorithms

3.1. MMseqs overview

MMseqs (Many-against-Many sequence searching) is a novel software suite for very fast protein sequence searches and clustering of huge protein sequence data sets, such as sets of predicted protein sequences or 6-frame-translated open reading frames (ORFs) from large metagenomics experiments.

MMseqs consists of three modules. The first two modules do the sequence search: (1) the **prefiltering module** computes the similarities between all sequences in one set with all sequences in the other based on a very fast and sensitive alignment-free metric, the sum of scores of similar k-mers; (2) the **alignment module** is an SSE2-accelerated Smith-Waterman-alignment of all sequences that pass a cut-off for the score in the first module. Both modules are highly parallelized in order to make use of all the cores of a multi-core computer. The first two modules are the core of MMseqs, taking most of the runtime and resources.

For the clustering, the sequences of a database are first compared all-against-all with the prefiltering module and all prefiltering matches are then checked in the alignment module. The third (3) **clustering module** orders sequence sets in clusters based on the similarity graph obtained from the comparison of the sequence set with itself in modules 1 and 2.

Overview of the MMseqs search and clustering workflows is given in Figure 3.1.

In addition to the possibility to run modules separately, MMseqs implements three workflows. Workflows offer the possibility to run the sequence search and clustering in one step, and run cascaded clustering and updating. Workflows have less possibilities of parameter tuning than running MMseqs modules separately, but they are more comfortable to use and offer additional functionalities in case of the cascaded clustering and updating.

The first, sequence search workflow, does a sequence search by running prefiltering and alignment module. The second, MMseqs clustering workflow, clusters the input databases by running prefiltering, alignment and clustering modules. Additionally, it offers the possibility to run cascaded clustering, which clusters the input database incrementally in multiple steps, increasing clustering sensitivity in each step. By postprocessing the results from the clustering iterations, it produces clustering results as if done in a single clustering step. The third workflow is an updating workflow. It takes a new version of the database and already done clustering of an older database version and updates the clustering by deleting removed sequences and adding new sequences to the existing clusters.

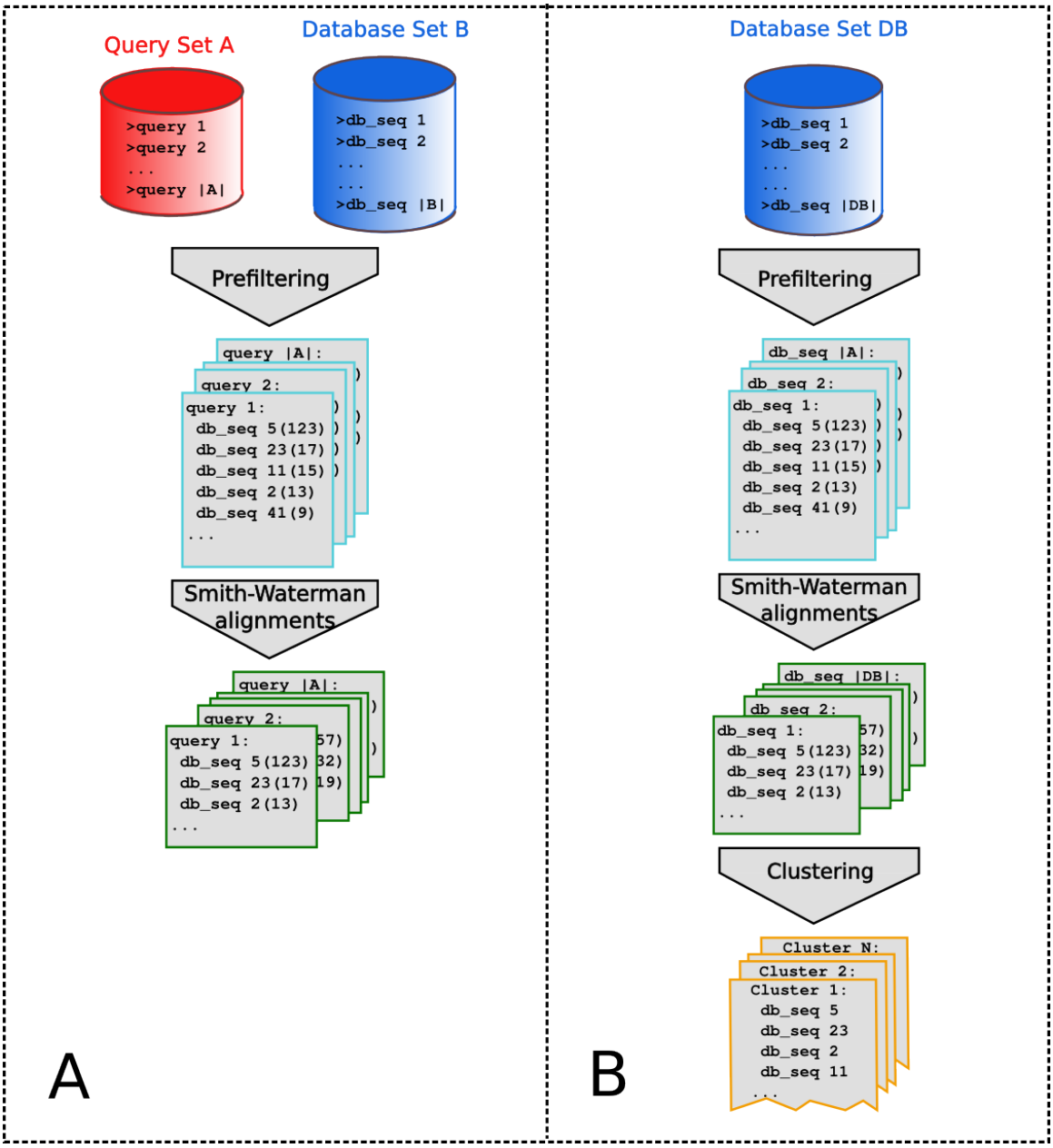


Figure 3.1.: **Overview over the two basic MMseqs workflows: sequence search and clustering. A: Sequence search.** All sequences from the query set A are matched against sequences from the database set B. Prefiltering produces for each sequence from set A a list of similar sequences from set B. In the alignment module, alignments are computed for all sequence pairs that passed the prefiltering. For each query sequence from set A, there is then a list of sequences from B producing significant alignments with the query sequence. **B: Clustering.** All sequences from the database are compared to each other in the prefiltering module and the prefiltering matches are checked in the alignment module. The result is a list of similar sequences together with the similarity scores and alignment statistics for each sequence in the database. Based on the alignment results, a clustering of B is defined.

The three modules of MMseqs and the workflows are explained in detail in the following sections.

3.2. Prefiltering

The prefiltering module is the core of MMseqs and is crucial for its speed and sensitivity. It does a fast and sensitive all-against-all comparison of the sequences contained in a query

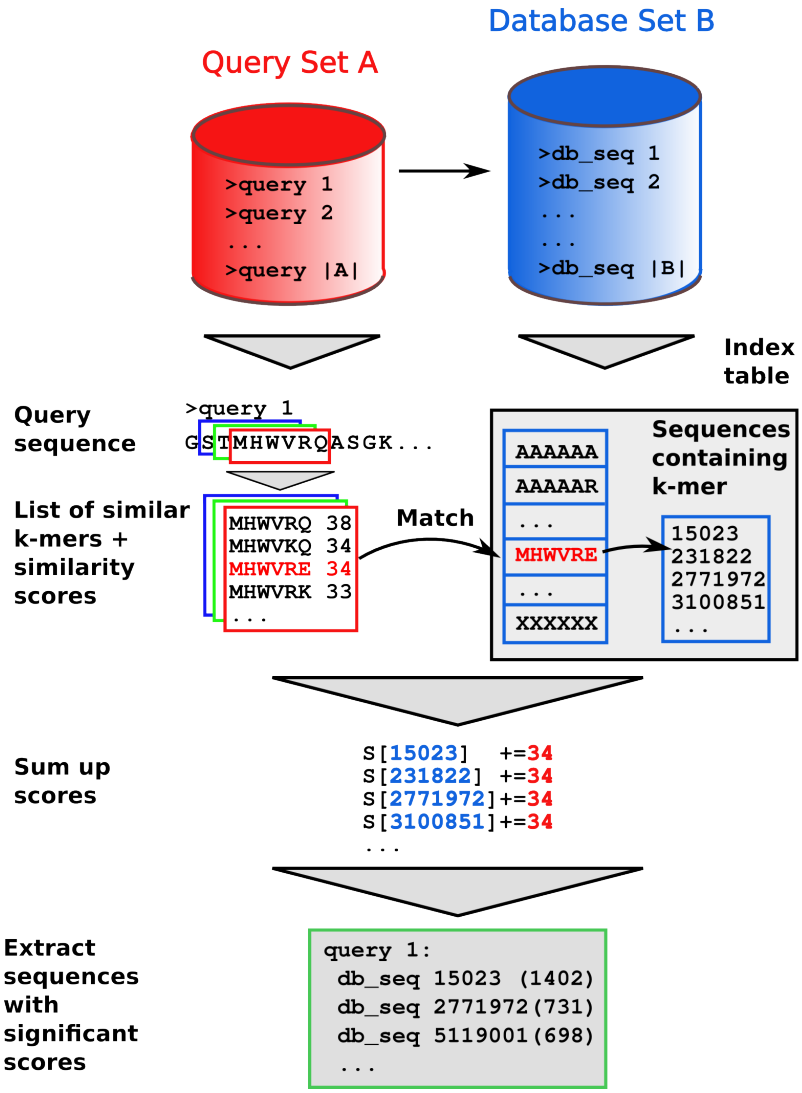


Figure 3.2.: **Overview over MMseqs prefiltering module.** All sequences from the query dataset A are compared to all sequences from the database B. First, all k-mers for every sequence in B are stored in an index table. Then prefiltering scores are calculated for each query sequence from A. For each k-mer in the query sequence, a list of similar k-mers is generated together with the similarity scores. For each k-mer in the list, a list of sequences in the database B containing this k-mer is retrieved from the index table. The similarity score is added to the overall score for the database sequence. After the query sequence is processed, database sequences producing a significant score are identified using a statistical approach and stored in the result list for the query sequence.

set A to the sequences in the database set B, using a very fast, alignment-free metric. The overview over the prefiltering module is shown in Figure 3.2. During the prefiltering, most of unrelated sequence pairs are eliminated, such that much slower exact sequence comparison has to compute pairwise alignments for only a very small fraction of all-against-all sequence pairs.

The main idea of the prefilter is the matching of short similar subsequences of a fixed length k, so-called k-mers, between the query and the database sequence. The similarity score of each k-mer pair is the BLOSUM score of its ungapped alignment. MMseqs prefilter identifies all k-mers with a similarity score above a threshold for each pair of sequences. The sum of the similarity scores of all matching k-mer pairs is the prefiltering score for the sequence pair. Sequences from the database B passing the prefiltering for a query sequence from A are identified basing on raw prefiltering scores using a statistical approach.

3.2.1. Similar vs. identical k-mers

Counting *similar* k-mer words for pairs of sequences is much more sensitive than counting *identical* k-mers, because it allows us to keep the word length k large while still maintaining a high sensitivity for detecting similar pairs of sequences at low sequence identities. A higher word length in turn reduces the number of chance k-mer matches much more than the number of k-mers matching as a result of the common ancestry of the two sequence segments. But MMseqs does not simply count similar k-mers, instead it sums up their BLOSUM62 similarity scores. This results in a further improvement in sensitivity, since k-mer pairs formed by chance will have lower scores on average than k-mer pairs that match due to their homology.

Consider first two homologous sequences with a sequence identity p_{seqid}, which we interpret as the probability that two homologous residues are identical. Assuming that the match probability of the k positions in a k-mer are approximately independent, the probability for two homologous k-mers to be identical is p_{seqid}^k. (In fact, since conserved positions are usually clustered in proteins, the true probability is actually larger than that, which will make the following estimates conservative.) For counting matches of similar k-mers, we demand that the score is larger than a certain threshold, such that for every k-mer there are on average r similar k-mers above this threshold (for example $r \approx 100$ in the 6-mer prefilter with default settings). The probability for two homologous k-mers to be similar is then approximately rp_{seqid}^k. To be able to detect the homology between two sequences, we need to count enough similar matching k-mers. If the sequences have a sequence identity p_{seqid} and their alignment has a length L, the number of expected matches should obey

$$rp_{\mathrm{seqid}}^k L \gg 1\,. \tag{3.1}$$

The probability for two random k-mers to be identical is p_{ran}^k, where p_{ran} is the probability

to observe two identical amino acids by chance. We estimate this probability using known background amino acid probabilities $p_{\mathrm{bg}}(a)$: $p_{ran} = \sum_{a=1}^{20} p_{\mathrm{bg}}(a)^2 \approx 0.058$. Hence, the probability for two random k-mers to have a similarity above the threshold is rp_{ran}^k. A second necessary condition to be able to distinguish two homologous sequences of length L with sequence identity p_{seqid} from other, non-homologous sequences is that the homologous pair yield many more similar k-mer pairs than the number of chance matches $rp_{\mathrm{ran}}^k L^2$ in the pair of non-homologous sequences. Therefore, we should have

$$\left(\frac{p_{\mathrm{seqid}}}{p_{\mathrm{ran}}}\right)^k \gg L. \tag{3.2}$$

For $p_{\mathrm{seqid}} = 0.3$, the term under the power of k is $p_{\mathrm{seqid}}/p_{\mathrm{ran}} \approx 5$. Therefore, we gain a factor of five for each of position of the k-mer in the ratio of k-mer matches between homologous over nonhomologous sequences. We therefore should choose k as large as possible. For $k = 6$ we obtain $5^6 \approx 15\,000 \gg L$, showing that for sequences longer than 15 000 residues, the number of chance 6-mer matches begins to outweigh the number of matches due to real homology. When using 7-mers, we get $5^7 \approx 78\,000 \gg L$, so there is basically no protein sequence that is long enough to produce enough chance 7-mer matches to outweigh the number of matches due to real homology. To estimate a suitable value r, note that eq. 3.1 tells us that we need to detect a sufficient number of matches even for short proteins. For a length $L = 50$ and $k = 6$, for example, the equation demands that $r \gg 1/(0.3^6 \times 100) \approx 40$. Hence $r = 100$ seems like a reasonable choice for the default similar 6-mer list length.

3.2.2. Index table generation

The index table guarantees a fast access to all database sequences containing a certain k-mer. In a preprocessing step of the prefiltering, before the prefiltering scores calculation starts, all k-mers of the database sequences are stored in the index table (Figure 3.3). For each possible k-mer, the index table stores a pointer to the list of all database sequences containing this k-mer.

The index table needs much memory, depending on k and the size of the database. The sizes of the sequence lists are distributed very unequally, so no average value can be assumed for the size of a sequence list. Therefore, the frequency of each k-mer is counted in the first pass though the database. Then, the memory for the sequence lists is allocated in one block to avoid memory fragmentation. The pointer array is then pointing to the position in the sequence lists block where the corresponding sequence list starts. The size of each sequence list is stored in an auxiliary array (Figure 3.3). Repeat k-mers are removed after the sequence list generation, i. e. each sequence is stored only once in the sequence list for every k-mer. For the distribution of the sequence list sizes for different k-mers, see Figure 3.4.

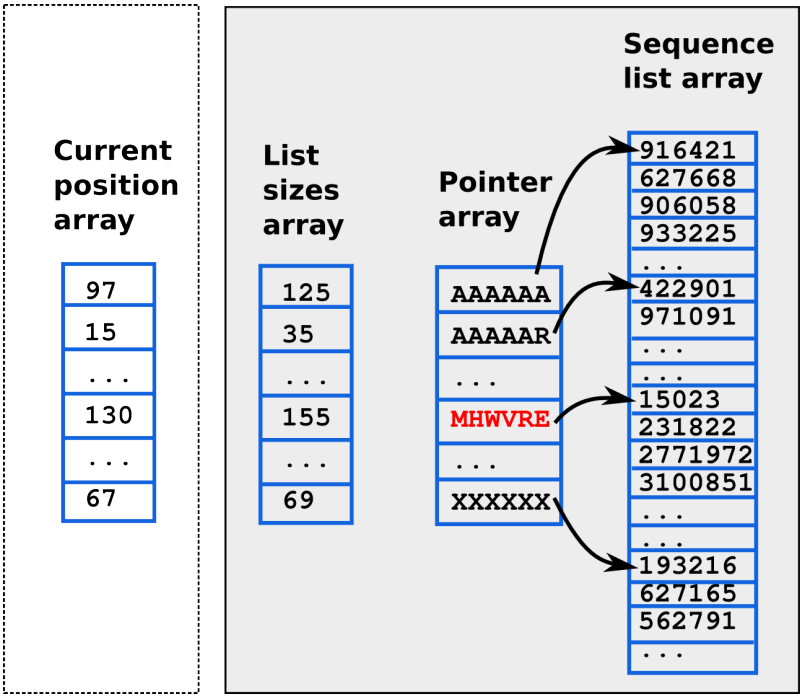


Figure 3.3.: **Index table.** The index table consists of three arrays (grey rectangle). The lists of database sequences containing a certain k-mer are stored consecutively for every k-mer in the sequence list array. The pointer array contains pointers to the beginning of the list of database sequences for every possible k-mer. The third list sizes array stores the size of every list. In addition, during the index table initialization a fourth array is necessary (current position array). It stores the current position in the sequence list and therefore, indicates the next position where the sequence ID of a sequence containing a k-mer can be written.

3.2.3. Memory consumption of the index table

The index table consists of the three arrays shown in the Figure 3.3: list sizes array, pointer array and sequence list array. One additional array is used during the filling of the index table which stores the current position in each k-mer sequence list where the next database sequence containing this k-mer can be written. The number of entries in this array is the same as the number of entries in the pointer array and in the list sizes array. This is the number of possible k-mers, which is a^k, where a is the alphabet size and k is the k-mer size. The current position array and the list sizes array contain unsigned integers (unsigned integer size is 4 byte), the pointer array contains pointers (pointer size in a 64-bit system is 8 byte). Therefore, these three arrays need $(4 + 4 + 8) \times a^k$ B of memory.

The memory consumption of the sequence lists can be calculated as following: each sequence of length L_i contains $L_i - k$ k-mers. We need 4 byte to store the sequence ID in the sequence list for a k-mer. For a database containing N sequences with an average length L, the index table needs approximately $4 \times N \times L$ B for the sequence lists.

Therefore, the general memory consumption of the index table can be calculated with the following formula (in byte):

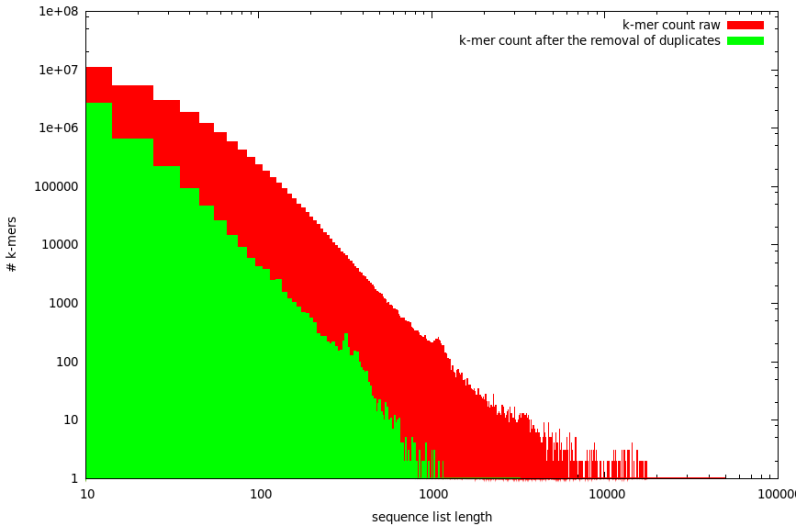


Figure 3.4.: **Distribution of the sequence list sizes in the index table for the different k-mers.** Sequence list size distribution is shown for the sequence list before and after the duplicates removal (red and green, respectively). The k-mer counts are divided into bins of size 10. Note the logarithmic scale on both axes. After the duplicates removal, there is much less k-mers with extremely large sequence list sizes. Such long lists occur mostly for frequently occuring repeat k-mers (like "AAAAAA") and by removing repeats within sequences, the length of the sequence list for such k-mers is drastically reduced.

$$M_{\text{index\_table}} \quad = \quad 16\,a^k + 4\,N\,L \tag{3.3}$$

Based on this formula, we can calculate the memory consumption of the index table for storing k-mers of the recent version of the UniProtKB database containing 54 M sequences, average sequence length is 350. For storing 6-mers of this database, we need $16 \times 21^6\,\text{B} + 4 \times 54 \times 10^6 \times 350\,\text{B} = 1{,}2\,\text{GB} + 70{,}4\,\text{GB} = 71{,}6\,\text{GB}$. When using 7-mers, the memory consumption for the three arrays (first term in the equation 3.3) rises to $16 \times 21^7\,\text{B} = 26{,}8\,\text{GB}$.

3.2.4. Index table matching

Prefiltering scores are calculated for each query sequence separately. A query sequence is processed from left to right. The current prefiltering score for each database sequence is stored in the array S (Figure 3.2). For each query sequence, $S[\cdot] = 0$ at the beginning of the query sequence processing. At each query sequence position, the k-mer at this position is retrieved and the list of similar k-mers down to a certain score cutoff is generated and stored together with the similarity scores. For each k-mer from this list, a list of database sequences containing this k-mer is retrieved from the index table. The similarity score is already known from the similar k-mer list generation and added to the prefiltering score of each database sequence in the sequence list. Then, the next k-mer of the query sequence

is processed. At the end of the query sequence, array S contains prefiltering scores of all database sequences for this query sequence.

3.2.5. Retrieving results for a query sequence

Prefiltering scores of the query with the most database sequences are 0, or very low, when composed only of chance k-mer matches between the query sequence and the database sequence. Truly homologous sequences have much higher prefiltering scores than the average score in the array S. However, this average score varies greatly among the different query sequences, depending on the length of the query sequence, the length of the matched database sequence and on their amino acid composition. Longer database sequences tend to have higher prefiltering scores with the query just because of chance matches. A rigid prefiltering score threshold for all sequences does not work well, missing homologous sequences in some cases and letting many unrelated sequences pass in other cases.

We do a statistical analysis of the distribution of the prefiltering scores for each query in order to find a query-specific prefiltering threshold for every query sequence. We assume that most of the database sequences are not homologous to the query sequence and cause only chance matches with it.

We are interested in a query-specific and database-sequence-specific prefiltering threshold above that the prefiltering score for the sequence pair becomes significant.

We correct for the fact that longer database sequences have a higher expected chance score. First, we calculate the sum of database sequence lengths:

$$\mathrm{sum}_L = \sum_{t=1}^{N} (L_t - k + 1)$$

and the sum of all prefiltering scores for a query sequence q with the database:

$$\mathrm{sum}_S = \sum_{t=1}^{N} S_{qt}$$

where N is the database size, L_t is the length of the database sequence t and S_{qt} is the prefiltering score of the query sequence q with a database sequence t. Then, the expected chance prefiltering score between q and t is

$$S_0 = (L_t - k + 1) \frac{\mathrm{sum_S}}{\mathrm{sum_L}}$$

We also correct for the lower score dispersion at high lengths. For this purpose, we need to determine the standard deviation of the score. We assume that the number of k-mer

matches is Poisson-distributed, so the standard deviation of the score should be proportional to the square root of the number of expected k-mer matches. We calculate the expected number of matches of q with the database. It is

$$n_{\text{match}} = (L_t - k + 1)\,\frac{\text{sum}_\text{S}}{\text{sum}_\text{L}}/S_{\text{match}}$$

where S_{match} is the expected score per chance k-mer match. Under the assumption that the number of k-mer matches is Poisson-distributed, the standard deviation of the chance score of the query sequence with a database sequence t is

$$\begin{aligned}\sigma_S &= \sqrt{n_{\text{match}}}\,S_{\text{match}} \\ &= \sqrt{(L_t - k + 1)\,\frac{\text{sum}_\text{S}}{\text{sum}_\text{L}}\,S_{\text{match}}}\end{aligned}$$

Therefore, the offset- and scale-corrected score Z_{qt} for a query sequence q and a database sequence t is

$$Z_{qt} = \frac{S_{qt} - S_0}{\sigma_S}$$

We are interested in all sequence pairs, where $Z_{qt} > Z_{thr}$. I. e. the prefiltering score should fulfill the condition

$$\begin{aligned}S_{qt} &\geq Z_{thr}\,\sigma_S + S_0 \\ &\geq Z_{thr}\sqrt{(L_t - k + 1)\,\frac{\text{sum}_\text{S}}{\text{sum}_\text{L}}S_{\text{match}}} + (L_t - k + 1)\,\frac{\text{sum}_\text{S}}{\text{sum}_\text{L}}\end{aligned}$$

However, calculating the offset- and scale-corrected score threshold for each pair q, t would slow down the retrieving of prefiltering results. Since the threshold value $Z_{thr}\,\sigma_S + S_0$ depends only on the length of the database sequence t for a fixed q, the database sequences are ordered by length, the threshold is calculated once and recalculated only if the length of the next sequence falls below 95% of the reference sequence length.

The statistical analysis of the scores gets unreliable for small databases, since in this case there exist not enough values to calculate the expected score and the standard deviation reliably. We use pseudo-counts to generate a reliable score threshold. We define the size of the pseudo-database as 100 000 with an average sequence length 350 (average sequence length in UniProtKB). We estimate k-mer match probability and set the score of a chance k-mer match S_{match} to be slightly above the k-mer similarity threshold. k-mer match probability estimation is explained in detail in section 3.2.10. Then, we calculate n_{match},

sum_S and sum_L with real values adding the pseudo-counts.

3.2.6. Parallelization

The prefiltering module is parallelized in two ways. First, it calculates on all the cores of the computer (or the number of cores specified by the user) by using OpenMP parallelization. Each core calculates prefiltering scores for a bunch of query sequences with the whole database and writes the results to a separate output database, one output database per core. After prefiltering scores calculation is complete, prefiltering results from each thread are merged into one output database (Figure 3.5).

In addition, the time-critical retrieving of the database sequences passing the prefiltering threshold for a query sequence is parallelized by utilization of vector processing units on the

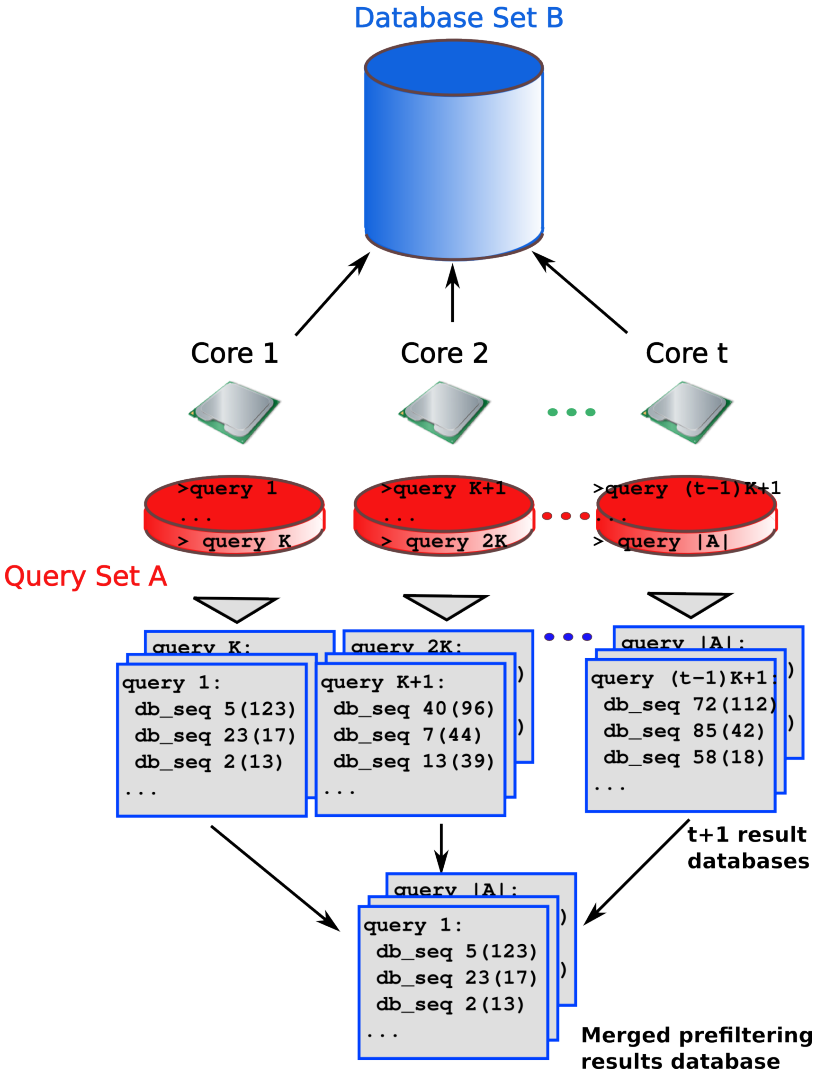


Figure 3.5.: **Parallelization with OpenMP.** Parts of the query sequence set are matched against the database in parallel. Each thread writes to its own output database. In the end, all results are merged into one output database.

CPU through SIMD instructions (SSE2 instruction set). By using SIMD parallelization, it is possible to perform an operation on a vector of values instead on a single value in one processor cycle.

For be able to use SSE2, we allocate aligned memory for the score array S (see Figure 3.2), so we can access the values in S vector-wise. Additionally, we store sequence specific prefiltering score thresholds for each database sequence in another array T, also using aligned memory allocation (for the threshold calculation, see the chapter 3.2.5). We need to compare $S[i]$ with $T[i]$ for each database sequence. Using SSE2 instructions, 8 checks are made in parallel on one core, therefore accelerating the result retrieving 8-fold.

3.2.7. Reduction of the memory consumption

Database sequence lists for each k-mer, stored in the index table, need a lot of memory for large databases. To hold all the sequence lists in the database for the UniProtKB containing 54 M sequences, we need approximately 70 GB of main memory (see chapter 3.2.3). We offer the user the possibility to reduce the memory usage at the cost of longer runtimes. The target database is split in multiple chunks and the prefiltering scores of the queries with the database are calculated only with one chunk at once. In the last step, the prefiltering result lists are merged (Figure 3.6).

3.2.8. Different k-mer sizes

MMseqs is able to use different k-mer sizes. In the last section, we stated that k-mer size should be as large as possible to make use of the enhanced signal-to-noise-ratio of larger k-

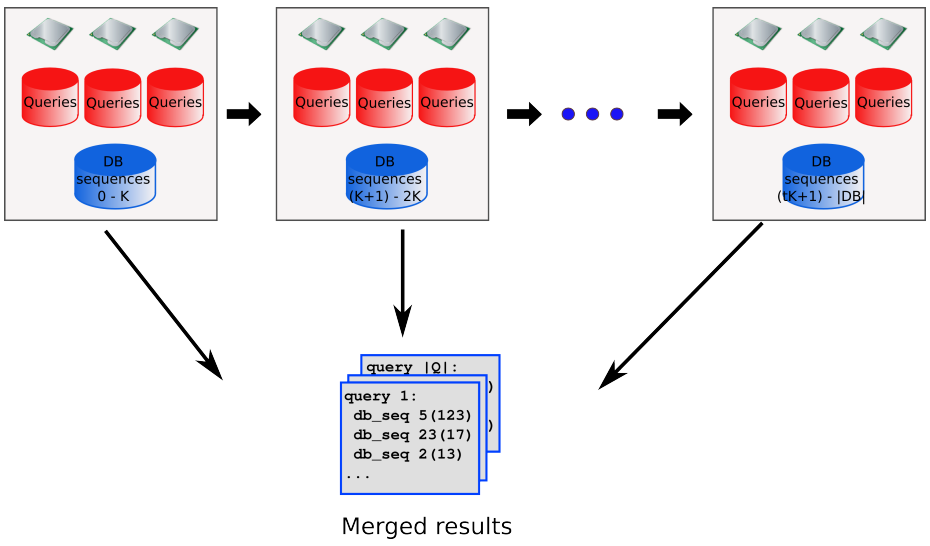


Figure 3.6.: **Reduction of memory consumption.** The target database (blue) is divided into $t+1$ chunks. In each step, matches of the query database to the current chunk of the target database is calculated in parallel using OpenMP parallelization (red). In the end, result lists from each step are appended to each other.

mers. In practice, k is limited by two considerations. First, the index table needs a memory of $21^k \times 16B$ for the index itself and for the auxiliary arrays. Therefore, the memory for the index table restricts the maximum k to 7 in the practical use. For $k = 6$, index table and the auxiliary arrays for k-mer counting need about 1,2 GB for the index and the auxiliary arrays in addition to the memory for the database sequence lists. For $k = 7$, this memory consumption increases to about 26 GB.

Second, larger k makes exponentially larger similar k-mer list lengths r necessary to achieve the same similarity threshold per k-mer position. This results in longer runtimes for k-mer lists generation and therefore, more frequent accesses to the lists of sequences for a k-mer in the index table. On the other side, the probability for a chance match is reduced by a factor p_{ran}, what makes prefiltering score addition and score retrieving faster for longer k-mers. Therefore, the usage of long k-mers pays off only if the score counting and retrieving predominates the running time. We want to check which database size is necessary for the score counting and the score retrieving to dominate the running time.

The time for the generation of the lists of similar k-mers T_1 can be described with the following equation:

$$T_1 \quad \propto \quad N_q \, L_q \, r \tag{3.4}$$

where N_q is the number of the query sequences, L_q the average length of a query sequence and r the average length of the similar k-mer lists. The time for the the score counting and retrieving can be described as follows:

$$T_2 \quad \propto \quad N_q \, L_q \, N_t \, L_t \, p_{\text{ran}}^k \tag{3.5}$$

where $N_t$and L_t are the number and the average length of the sequences in the target database and p_{ran}^k describes the probability of a chance k-mer match, as stated in the previous section. For T_2 to dominate the running time, $\frac{T_2}{T_1} > 10$ should apply. I. e.,

$$\begin{aligned} \frac{T_1}{T_2} \quad &\propto \quad N_t \, L_t \, \frac{p_{\text{ran}}^k}{r} \\ &\propto \quad N_t \, L_t \, p_{\text{ran}}^k > 10 \end{aligned}$$

Subsequently, the minimum number of the database sequences for the the score counting and retrieving to predominate the running time should fulfill the following condition:

$$N_t \quad > \quad \frac{10}{L_t \, p_{\text{ran}}^k}$$

The average length of the sequences in the UniProtKB database is approximately 350, and $p_{ran} \approx 0.058$ as stated in the section 3.2.1. It follows that the minimum number of the sequences in the database for 6-mers to pay off versus the use of shorter k-mers is $\frac{10}{350 \times 0.058^6} \approx 750\,000$, whereas the number of the sequences in the database for 7-mers to pay off is $\frac{10}{350 \times 0.058^7} \approx 12\,000\,000$. So, the usage of 7-mers pays off only for very large databases containing more than 12 M sequences.

3.2.9. Reduced amino acid alphabet

The user has the option to switch to a reduced amino acid alphabet. A reduced amino acid alphabet is generated by subsequently merging two most similar amino acids. The similarity is assessed by using the mutual information of their substitution probability distributions. The merging process is continued until a desired reduced alphabet size is reached.

Using an amino acid alphabet size reduced by 4 reduces the memory needed for indexing 7-mers from 26,8 GB to 6,1 GB (first term in the equation 3.3), without considering the memory for the sequence lists, since it does not depend on k-mers size but only on the number of residues in the database.

Mutual information is the measure of two random variables' mutual dependence. Mutual information for a pair of amino acids is calculated based on the substitution and the background probabilities of the amino acids taken from the substitution matrix. The goal is to merge a pair of amino acids that result in the least mutual information loss in the amino acid substitution matrix. For this purpose, we combine each possible amino acid pair $(i,j), i \neq j$ into one merged amino acid and calculate the mutual information of the resulting matrix.

For each amino acid i, we know the background probability $p_{back}(i)$, and for each amino acid pair (i,j) we know the probability $p(i,j)$ that i is aligned with j. Based on p_{back} and p, an amino acid substitution matrix is calculated as following:

$$S(i,j) \quad = \quad \log \frac{p(i,j)}{p_{back}(i)\, p_{back}(j)}$$

In each step, we merge an amino acid pair $(i,j), i \neq j$ into one new (imaginary) merged amino acid m and calculate its background probability:

$$p_{back}(m) \quad = \quad p_{back}(i) + p_{back}(j)$$

and the new alignment probabilities $p(m,\cdot)$ of m with other amino acids in the amino acid alphabet:

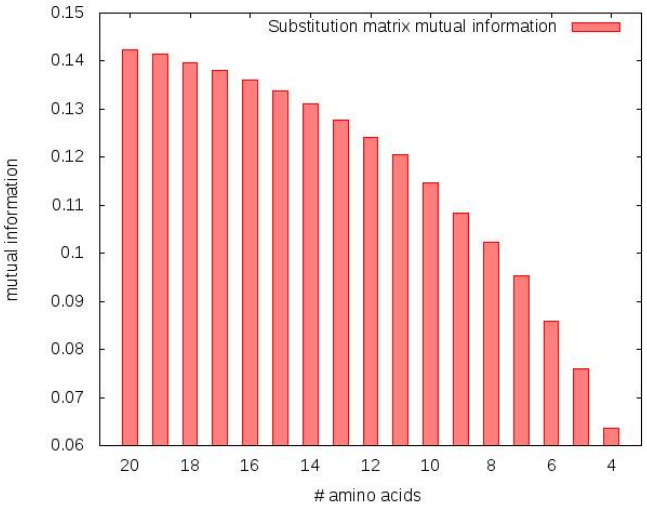


Figure 3.7.: **Mutual information of the amino acid substitution matrix with reduced amino acid alphabets.** In each step, most similar amino acids are merged and the mutual information for the resulting reduced alphabet is calculated. The size of the amino acid alphabet is shown on the x-axis, mutual information on the y-axis.

$$p(m,\cdot) \quad = \quad p(i,\cdot) + p(j,\cdot)$$

Then, a new substitution probability matrix S_m is generated for the reduced amino acid alphabet based on new distributions $p_{back}$and p. Finally, the mutual information I_m of the resulting substitution matrix is measured:

$$\mathrm{I_m} = \sum_{\mathrm{i}} \sum_{\mathrm{j}} \; p(i,j) \; \log \frac{p(i,j)}{p_{back}(i)\, p_{back}(j)}$$

The process of merging an amino acid pair and calculating the resulting mutual information is repeated for each amino acid pair. Finally, the amino acid pair with the highest value of I_m is chosen and merged into a new amino acid, i. e. the amino acid alphabet is reduced by one.

The whole process is repeated until a target number of amino acids is reached.

Figure 3.7 shows the mutual information for the different sizes of the reduced amino acids alphabets, based on the BLOSUM62 matrix.

3.2.10. Automatic sensitivity setting

Using different k-mer sizes and different average similar k-mer list lengths, it is possible to run MMseqs with different sensitivity. Longer similar k-mer lists per position make MMseqs more sensitive but also slower. Shorter similar k-mer lists reduce the sensitivity

and increase the speed. Again, longer k-mers need longer k-mer lists to be as sensitive as shorter k-mers. MMseqs makes different settings comparable by adjusting the k-mer similarity threshold and ensuring that the prefiltering run takes the same time for the same sensitivity parameter value despite the different settings.

The most time-consuming parts of the prefiltering module is the generation of the similar k-mer lists and the index table matching. Therefore, for a fixed database size the runtime can be described with the following expression (cf. eq. 3.4 and eq. 3.5 in section 3.2.8):

$$\text{runtime} \quad \propto \quad \alpha\, r + \beta\, p_{\text{match}} + \gamma \tag{3.6}$$

where r is the average length of the similar k-mer list per query sequence position and $p_{\text{match}}$is the probability of one query sequence position to match one database sequence position. For the purpose of the usability, we set

$$2^{\text{runtime}} \quad = \quad \alpha\, r + \beta\, p_{\text{match}} + \gamma$$

so increasing the variable runtime by one doubles the runtime approximately and increases the sensitivity of the prefiltering allowing longer similar k-mer lists to be generated and examined.

I ran MMseqs with the different k-mer lengths and alphabet sizes and fitted the parameters α, β and γ by measuring the runtime and the corresponding values of $L_{\text{kmer\_list}}$ and p_{match}.

Subsequently, we are able to compare the runtime of the different settings by measuring r and p_{match} and calculating 2^{runtime}. During the initialization phase, MMseqs does a couple of test prefiltering runs using only a small fraction of the data with the different k-mer similarity thresholds and records r and p_{match}. It adjusts the k-mer similarity threshold and therefore changes r and p_{match} until the left-side expression 2^{runtime} matches a specified value of runtime.

To be able to do a fine tuning of the k-mer list lengths and therefore of the sensitivity, we set the score unit for the k-mer similarity threshold that controls the lengths of the similar k-mer lists to 1/8 bit. After the generation of the similar k-mer lists, we switch the score to 1/2 bit scale to avoid the frequent overflow of the unsigned short integer range in the array $S[\cdot]$ during the prefiltering scores calculation (see section 3.2.4).

Similar to the statistical prefiltering threshold calculation, we have to use pseudocounts if the database is too small to measure p_{match} reliably:

$$p_{\text{match}} \quad = \quad \frac{n_{\text{matches}} + n_{\text{matches\_pc}}}{\sum L + \sum L_{pc}}$$

where n_{matches} is the observed number of matches of the query and $\sum L$ is the sum of lengths of the database sequences. $n_{\text{matches\_pc}}$ and $\sum L_{pc}$ are the pseudocount contributions to the number of matches and the lengths.

Pseudocount components are calculated as follows: the pseudo database size is set 1000 query sequences and 100 000 database sequences with an average length 350, resulting in the pseudo-count contribution $\sum L_{pc} = 1000 \times 350 \times 100\,000 \times 350$, since chance matches can occur anywhere between the two sequences. The probability of a k-mer to match another k-mer by chance is set to $\left(\frac{1}{a}\right)^k$, where a is a size of the amino acid alphabet. This is a little lower than would be calculated based on BLOSUM62 self-substitution probabilities, but matches the observed values better. Pseudocount number of database matches is then

$$n_{\text{matches\_pc}} = \sum L_{pc}\, r \left(\frac{1}{a}\right)^k$$

Dependence of the running time on the database size It is necessary to note that the preceding sensitivity setting is only valid for a fixed database size. Equation 3.6 generalizes equations 3.4 and 3.5. Based on the equations, the full equation for the runtime is

$$\text{runtime} \propto N_q\, L_q\, r + N_q\, L_q\, N_t\, L_t\, p_{\text{match}} + \gamma \tag{3.7}$$

$$\propto r + N_t\, L_t\, p_{\text{match}} + \gamma \tag{3.8}$$

Therefore, a particular fitting of the sensitivity and runtime is valid only for a fixed database size and should be adapted if the database size changes considerably.

3.2.11. Amino acid local composition bias correction

Some sequences have regions of low complexity with an amino acid composition that differs considerably from the background amino acid distribution assumed in the amino acid substitution matrix. Therefore, low complexity regions of a sequence can cause high prefiltering scores with low complexity regions of other sequences simply because of the same local amino acid composition bias but not for the reason of true homology. To alleviate this effect, we correct for the local compositional bias in the sequences by assigning lower scores to the matches of frequent amino acids. We examine d amino acids on both sides of the the amino acid x_i at position i in the sequence. Score correction ΔS_i at position i is

$$\Delta S_i(x_i) = -\frac{1}{2d} \sum_{j=i-d, j\neq i}^{i+d} S(x_i,x_j) + \sum_{a=1}^{20} f(a) S(a,x_i)$$

where $S(x_i,x_j)$ is the amino acid substitution score between amino acids x_i and x_j, and

$f(a)$ is the background frequency of the amino acid a. Then, the final corrected score S_c for the match of x_i with another amino acid y_j is

$$S_c(x_i,y_j) \quad = \quad S(x_i,y_j) + \Delta S_i(x_i)$$

Therefore, amino acids that are less frequent in the window $\pm d$ around the sequence position i than the background frequency of this amino acid contribute more to the score, and the more frequent less.

3.3. Alignment module

The alignment module calculates Smith-Waterman alignments for sequence pairs that passed the prefiltering. It implements a SSE2 vectorized, OpenMP parallelized Smith-Waterman alignment with Gotoh improvements for handling affine gap penalties. It is based on the implementation by Michael Farrar. For the detailed explanation of the Farrar Smith-Waterman alignment implementation, see the section 1.3.2.

Farrar first stores 16 8-bit integers in the SIMD registers and recalculates the score using 8 16-bit integers if it exceeds the maximum value of 255 that can be stored in an 8-bit integer. The first calculation is twice as fast as the second. Farrar assumes here that most sequence pairs are not homologous and therefore get very low scores in the alignment and don't have to be recalculated. However, sequence pairs passing the prefiltering module in MMseqs are mostly homologous. Between 60% and 99% of the sequences passing the prefilterings module also pass the alignment module, depending on the settings. The calculation of the Smith-Waterman scores using 8-bit integers only slows down the calculation needlessly, since most scores have to be recalculated anyway. Therefore, we calculate the scores storing 8 16-bit integers in the SIMD registers right in the first pass.

In addition to the SIMD parallelization, the alignment module is parallelized with OpenMP similarly to the prefiltering module. Each core calculates one alignment at one point of time. Each thread has its own temporary output database, so the output does not have to be synchronized. After the calculation is complete, the outputs are merged into one output database.

3.3.1. Backtracing

The original Farrar implementation only outputs the alignment score for a sequence pair. However, we also need the alignment statistics such as alignment coverage and the sequence identity. Therefore, I implemented the backtracing of the alignment. Pseudocode is shown in the Algorithm 1. For the backtracing, we store the three dynamic programming matrices and the position of the maximum score during the score calculation. Then we start at

the maximum position and do a traceback of the alignment backwards, stating the next alignment step - match or gap in either direction - based on the match, gap open and gap extension scores and the score recorded in the dynamic programming matrices at the respective position. During the backtracing, the length of the alignment and the number of matches is recorded.

Traceback stops if it reaches a dynamic programming matrix cell with the score 0. Then, the end position of the alignment is recorded and the alignment coverage of both query and database sequence and the sequence identity of the alignment are calculated. Additionally, an e-value is calculated using Karlin-Altschul statistics for local alignments (see chapter 1.5.2).

Algorithm 1 Smith-Waterman, Gotoh alignment traceback.

```
1: i = maxScorePosQuery
2: j = maxScorePosDbSeq
3: while i > 0 && j > 0 do          ▷ i: position in the query, j: position in the database sequence
4:     if H[i][j] == H[i-1][j-1]+ score(q[i], db[j]) then      ▷ match between query and db sequence
5:         i = i-1
6:         j = j-1
7:     else if H[i][j] == 0 then
8:         break
9:     else if H[i][j] == E[i][j] then      ▷ continue with the E matrix, i.e. gap in the db sequence
10:        while i > 0 && j > 0 do
11:            if E[i][j] == E[i][j-1] - gap_extend then
12:                j = j-1
13:            else if E[i][j] == H[i][j-1] - gap_open then
14:                j = j-1
15:                break                                   ▷ end of the gap - leave the E matrix
16:            end if
17:        end while
18:    else if H[i][j] == F[i][j] then  ▷ continue with the F matrix, i.e. gap in the query sequence
19:        while i > 0 && j > 0 do
20:            if F[i][j] == F[i-1][j] - gap_extend then
21:                i = i-1
22:            else if F[i][j] == F[i-1][j] - gap_open then
23:                i = i-1
24:                break                                   ▷ end of the gap - leave the F matrix
25:            end if
26:        end while
27:    end if
28: end while
29: return i,j                                  ▷ return the end positions of the alignment
```

3.3.2. Memory consumption

Storing the dynamic programming matrices for the traceback implies a higher memory consumption than in the original Farrar implementation. Originally, only the previous and the current columns of the dynamic programming matrix are stored. To make a traceback

possible, we have to store all the three dynamic programming matrices completely. For aligning two sequences with lengths L_i and L_j, each core needs $3 \times L_i \times L_j \times 16$ bits of memory. In the worst case, assuming the maximum sequence length of about 36 000 residues, the memory consumption for the alignment calculation is about 7,5 GB for the three dynamic programming matrices. For the average UniProtKB sequence length of 350, the memory consumption is less than 1 MB.

We want to keep the memory usage at a reasonable level and at the same time avoid slowing down the program unnecessary. Allocating memory only once would mean that we need to consider the memory required to aligne the longest sequences, i. e. for a computer with 32 cores, we would need $7{,}5 \times 32 = 240$ GB memory, what is clearly not feasible. On the other side, re-allocating memory for each sequence pair would consume not more memory than necessary but allocating and deleting memory after each alignment is computationally expensive.

I implemented a compromise of these two solutions. The memory for the three dynamic programming matrices is allocated when the program starts for sequences with a maximum length of 1000, i. e. 5,7 MB per core. When the length of the query or the database sequence exceeds 1000, the memory for the alignment is allocated separately and freed after the alignment calculation. Otherwise, already allocated memory is reused.

3.4. Clustering module

The clustering module takes the alignment results produced in the alignment module as input. The alignments in fact form an undirected sequence relationship graph, where the sequences are the vertices and the alignments are the edges.

We have implemented two clustering algorithms: the greedy set cover algorithm and the greedy incremental clustering algorithm as used by a lot of clustering tools, e. g. CD-HIT, kClust and USEARCH.

3.4.1. Greedy set cover

The algorithm transforms the Smith-Waterman results into an undirected sequence relationship graph. We try to select a minimal amount of vertices to cover the whole graph. This problem can be expressed a NP-complete optimization problem called set cover.

Chvatal at al (Chvatal, 1979) introduced an approximation for this problem, called greedy set cover. We implemented this algorithm to subdivide the graph into clusters by selecting an approximately minimal amount of vertices covering the whole graph. The Figure 3.8 illustrates the clustering with greedy set cover.

For our purpose, the algorithm should be fast enough to handle 10^8 vertices with about 50 connections per vertex. Our implementation of the algorithm has linear runtime and space complexity.

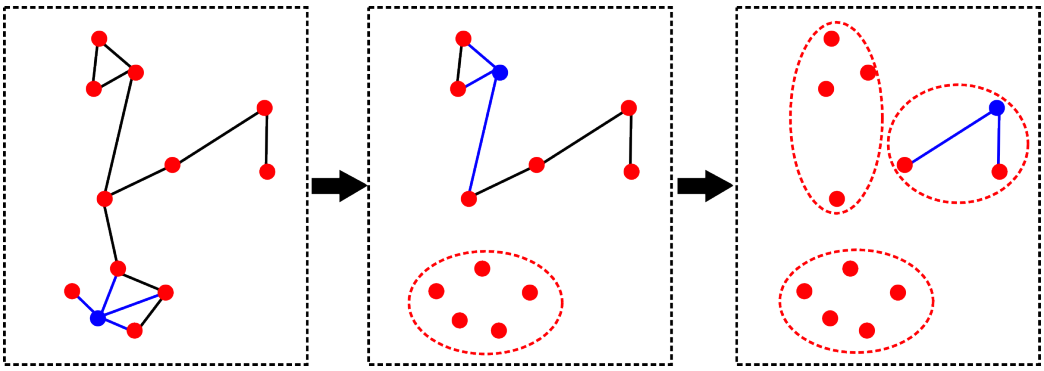


Figure 3.8.: **Set cover clustering algorithm.** In each step, the sequence with the most connections is chosen, i. e. in our case, the sequences with the most alignment results satisfying the search criteria (blue point, connections are blue lines). This sequence and the connected sequences are put into a cluster. All connections of the cluster members are deleted (image in the middle, dashed circle). In the next step, the sequence with the most connections is chosen from the still unclustered part of the database (image in the middle, blue point and blue lines). This is repeated until all sequences of the database are part of a cluster.

Before the clustering starts, the graph resulting from the run of the alignment module is stored in a data structure allowing efficient access to the vertices and edges of the graph. Every sequence is a vertex, every significant alignment is an edge. For every vertex, a set is created, containing a pointer to this vertex and also pointers to all vertices it has edges to. In addition, every vertex has pointers to all sets where it is contained. Eventually, an array lookup `A` containing pointers to the sets ordered by size is created. At each position i, it stores a pointer to the list of sets of size i.

For the pseudocode of the algorithm, see Algorithm 2. The algorithm iterates over `A` in descending order of the set sizes. A set containing the most sequences at this point (`set_max`) is selected within the loop. This can be done efficiently because we keep the sets ordered by size in an array lookup. All edges of the vertices in `set_max` are removed from other sets by following the pointers of each vertex to the sets containing this vertex . For each of these sets, the size of the set and the resulting position in the array `A` is updated (`remove_from_sets` function in Algorithm 2). The vertices contained in `set_max` form the next cluster and are added to the resulting clustering (`add` function). Then, the algorithm proceeds with the largest remaining set.

The algorithm has a linear runtime, since $\sum |\text{set\_max}|$ in the inner loop is equal to the size of the database.

3.4.2. Greedy incremental clustering

In addition to the clustering with set cover, also a simple greedy clustering algorithm is implemented, similar to CD-HIT and kClust clustering algorithm. It starts with the longest sequence and puts all the sequences from the alignment list of the longest sequence into the first cluster. All the sequences in this cluster are removed from the sequences to cluster

Algorithm 2 Greedy set cover.

```
1: result = empty_set
2: pos = size(A)
3: while pos > 0 do
4:     if A[pos] ≠ NULL then
5:         set_list = A[pos]
6:         for set_max ∈ set_list do
7:             remove_from_sets(set_max) // O(|set_max|)
8:             add(result, set_max)
9:         end for
10:    end if
11:    pos = pos - 1
12: end while
13: return result
```

and from their alignment lists. Then, the algorithm goes on with the longest remaining sequence until each sequence is assigned to a cluster.

The implementation of the algorithm is very similar to the implementation of the greedy set cover. Array A now contains sets ordered not by their size, but by the length of the representative sequence of the set. The representative sequence is the query sequence of the alignment list, other sequences in the set are the sequences of the database matched by the query sequence in the alignment module. The function `remove_from_sets` in the Algorithm 2 has only to remove the vertices contained in `set_max` from other sets. Pointers in A always remain the same, since the lengths of the representative sequences do not change and every sequence is only once a representative sequence of a set.

3.5. MMseqs workflows

In addition to the base modules, MMseqs offers three workflows that make the clustering easier and more efficient. The search workflow and the simple clustering workflow implement the simple sequence of MMseqs module calls necessary for sequence search or clustering, i. e. prefiltering and alignment modules for the search and prefiltering, alignment and clustering modules for the clustering. Cascaded clustering and update modules offer additional functionalities that cannot simply be reproduced by calling MMseqs prefiltering, alignment and clustering modules.

3.5.1. Sequence search workflow

Sequence search workflow runs the prefiltering and the alignment modules subsequently with predefined parameters. The most important parameters like search sensitivity still can be set by the user. The search workflow outputs a list of database sequences producing significant alignments for each query sequence (see Figure 3.1).

3.5.2. Clustering and cascaded clustering workflow

This workflow does a simple clustering of an input database in a single run. Internally, it executes the prefiltering, alignment and clustering modules consecutively with a user defined sensitivity (Figure 3.1). It makes it easy for a user to calculate a basic clustering of a protein sequence database. More advanced user still can execute the single modules consecutively what allows more sophisticated parameter tuning.

This workflow also offers the possibility to run a cascaded instead of simple clustering workflow. Cascaded clustering workflow clusters the input database incrementally by increasing the clustering sensitivity in each step. It combines three independent clustering steps. The first clustering step clusters the input database with the lowest possible sensitivity very fast. Then, only representative sequences of the resulting clusters are clustered in the next clustering step with higher sensitivity. Finally, representative sequences remaining after the second step are clustered with the user defined target sensitivity. By postprocessing the results from all clustering iterations, cascaded clustering produces clustering results as if done in a single clustering step. The whole process is illustrated in Figure 3.9.

Cascaded clustering yields more sensitive clustering at a constant speed since many homologous pairs are eliminated from the database early in a faster run with a low sensitivity. Particularly for large databases it is highly recommended to use the cascaded instead of the simple clustering workflow. For larger databases, the maximum cluster size is limited by two factors when running a simple clustering workflow. First, the amount of prefiltering results per query is limited to prevent the prefiltering from forming very long prefiltering lists for one sequence during the clustering of a large database. Otherwise, clustering results for a large database like UniProtKB clustered with a high sensitivity can easily need several TB of disc memory. Second, the alignment module would have to compute a very large amount of alignments for certain sequences, what can get very slow for large databases. Finally, the clustering module easily runs out of memory if it has to process a large amount of connections per sequence (in the order of a few hundreds). The default maximum cluster size is 50 for large databases containing tens of millions of sequences. Larger clusters result in very high consumption of resources.

The solution for the cluster size restriction is the cascaded clustering. In the first step, it clusters the database with the lowest possible sensitivity, therefore being very fast, and restricts the maximum result list length in all modules to 50, therefore avoiding high disc space consumption in the prefiltering module, long running times in the alignment module and high memory consumption in the clustering module. The first cascaded clustering step reduces the size of the UniProtKB database by the factor of two in the cascaded clustering run with default settings. In the second step, the sensitivity and the maximum result list length are raised, and the database size is reduced further. In the third step, the size of the database is reduced to a small fraction of the original size (e. g. approximately one fifth for cascaded clustering of the UniProtKB with default settings). The clustering is now

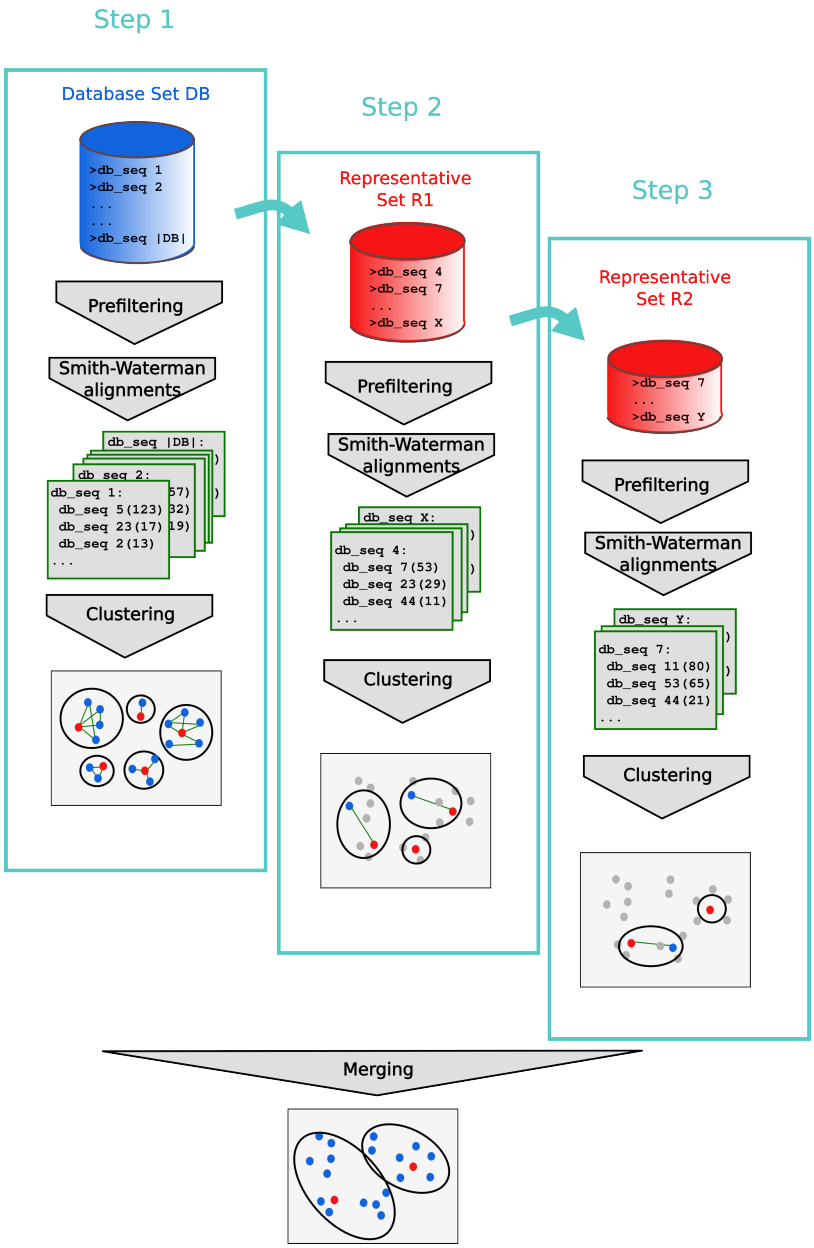


Figure 3.9.: **Cascaded clustering workflow.** Cascaded clustering workflow consists of three steps (cyan rectangles). In the first step, the database (blue) is clustered in the usual MMseqs workflow with the lowest possible sensitivity parameter value 1. The representatives of the clusters (red dots) yield a new input database (database R1 in red). This database is clustered with a fair sensitivity. The representatives of the resulting clustering (database R2) are clustered with high sensitivity. In the end, the clusters are merged based on the clustering assignments in each step (clustering result on the bottom, resulting in two clusters containing all the sequences of the initial database).

run with the maximum sensitivity and the longest result lists - the disc space and memory consumption and the running times are not problematic anymore with the smaller database size.

The maximum cluster size resulting from cascaded clustering increases by orders of magnitude: representative sequences from the previous run can accumulate sequences in the cluster in the next run and so, clusters can grow exponentially. With default settings, the maximum possible cluster size when using cascaded clustering is $50 \times 100 \times 300 = 1\,500\,000$. Since the clusters are merged only in the end of the calculation and all the already clustered sequences are not considered anymore in the subsequent clustering step, the space, memory and runtime consumption remain constant or even decrease compared to the straightforward clustering.

3.5.3. Updating workflow

Today, the amount of the protein sequence data is growing very fast. A large sequence database like UniProtKB grows exponentially (Figure 2.1a) and can easily accumulate several hundred thousand sequences per week. Sequences can also be deleted from the database. MMseqs updating workflow allows the user to add new sequences to and delete sequences from an existing clustering without the need to cluster the whole database from scratch. Updating workflow is illustrated in Figure 3.10.

MMseqs gets the old and the current versions of the database and the clustering of the old database version. Removed sequences are then deleted from the clustering. New sequences are either added to existing clusters if there is a significant match to a member of this cluster, or they form new clusters. The updating is very fast compared to the clustering from scratch, since much less similarities of sequence pairs have to be computed. Therefore, updating allows to keep the clustering of very large databases up to date at very low computational cost. Cluster identifiers remain stable during the updating process.

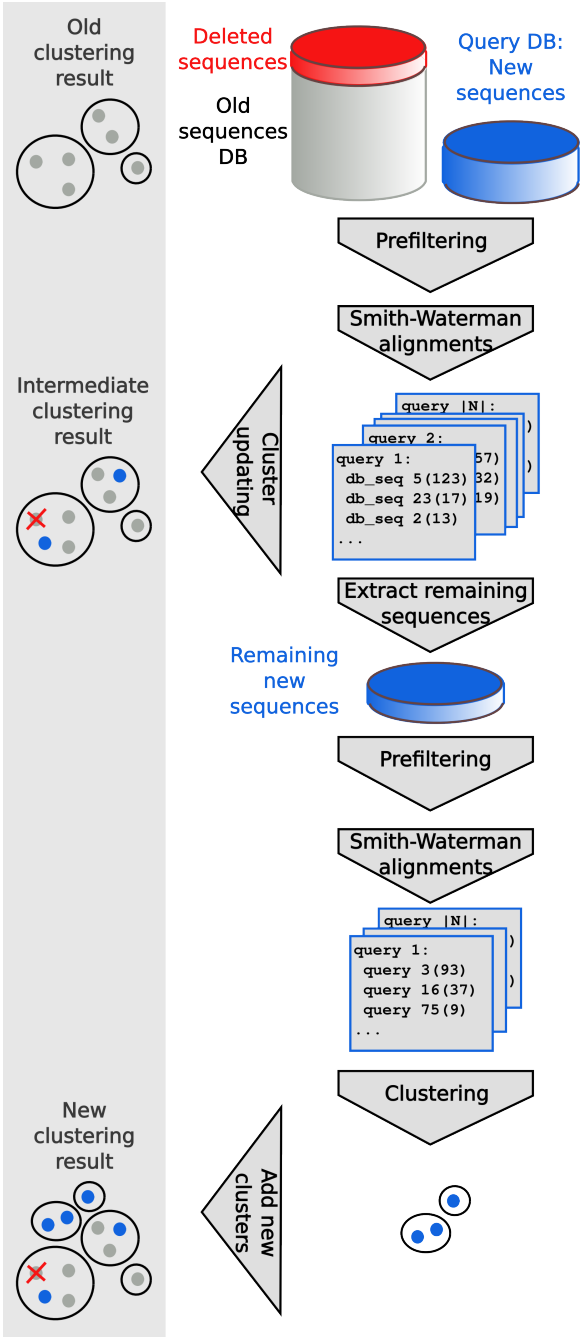


Figure 3.10.: **Updating workflow.** Input for the updating workflow is the old database version and its clustering. In the first step, sequences deleted from the database are determined (red upper part of the database) and deleted from clustering (red cross in the intermediate clustering). Then, the prefiltering and alignment modules determine new sequences (small blue database) that are similar to some sequences in the old database version. Sequences with matches to the old database are added to the corresponding clusters (blue dots in the intermediate clustering result). The rest of new sequences are clustered and the resulting clusters are added to the complete clustering.

4. MMseqs user guide

MMseqs suite for fast and sensitive batch searching and clustering of huge protein sequence sets

4.1. Summary

MMseqs (Many-against-Many sequence searching) is a software suite for very fast protein sequence searches and clustering of huge protein sequence data sets. MMseqs is around 1000 times faster than protein BLAST and sensitive enough to capture similarities down to less than 30% sequence identity.

At the core of MMseqs are two modules for the comparison of two sequence sets with each other - the prefiltering and the alignment modules. The first, prefiltering module computes the similarities between all sequences in one set with all sequences in the other based on a very fast and sensitive alignment-free metric, the sum of scores of similar k-mers. The alignment module implements an SSE2-accelerated Smith-Waterman-alignment of all sequences that pass a cut-off for the prefiltering score in the first module. Both modules are parallelized to use all cores of a computer to full capacity. Due to its unparalleled combination of speed and sensitivity, searches of all predicted ORFs in large metagenomics data sets through the entire UniProtKB or NCBI-NR databases are feasible. This could allow for assigning to functional clusters and taxonomic clades many reads that are too diverged to be mappable by current software.

MMseqs' clustering module can cluster sequence sets efficiently into groups of similar sequences. It takes as input the similarity graph obtained from the comparison of the sequence set with itself in the prefiltering and alignment modules. MMseqs further supports an updating mode in which sequences can be added to an existing clustering with stable cluster identifiers and without the need to recluster the entire sequence set. We will use MMseqs to regularly update versions of the UniProtKB database clustered down to 20-30% sequence similarity threshold.

4.2. Installation

First, set environment variables:

```
$ export MMDIR=$HOME/path/to/mmseqs/
```

```
$ export PATH=$PATH:$MMDIR/bin
```

MMseqs uses ffindex, a fast and simple database for wrapping and accessing huge amounts of small files. Setting the environment variable `LD_LIBRARY_PATH` ensures that ffindex binaries are in the path:

```
$ export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:$MMDIR/lib/ffindex/src
```

Then create MMseqs binaries:

```
$ cd $MMDIR/src
$ make
```

MMseqs binaries are now located in `$MMDIR/bin.`

4.3. Getting started

Here we explain how to run a search for matches of sequences in the query database in the target database and how to cluster a database. Test data (a query and a target database for the sequence search and a database for the clustering) are stored in `$MMDIR/data.`

Search

You can use the query database `queryDB.fasta` and target database `targetDB.fasta` to test the search workflow.

Before clustering, you need to convert your database containing query sequences (`queryDB.fasta`) and your target database (`targetDB.fasta`) into ffindex format:

```
$ fasta2ffindex queryDB.fasta queryDB
$ fasta2ffindex targetDB.fasta targetDB
```

It generates ffindex database files, e. g. `queryDB` and ffindex index file `queryDB.index` from `queryDB.fasta`. Then, generate a directory for tmp files:

```
$ mkdir tmp
```

Please ensure that in case of large input databases `tmp` provides enough free space. For the disc space requirements, see the section

To run the search, type:

```
$ mmseqs_search queryDB targetDB outDB tmp
```

Then, convert the result ffindex database into a FASTA formatted database:

```
$ ffindex2fasta outDB outDB.fasta
```

Clustering

Before clustering, convert your FASTA database into ffindex format:

```
$ fasta2ffindex DB.fasta DB
```

Then, generate a directory for tmp files:

```
$ mkdir tmp
```

Run the cascaded clustering of your database and output the result into the database files `DB_clu`, `DB_clu.index`:

```
$ mmseqs_cluster DB DB_clu tmp --cascaded
```

To generate a FASTA-style formatted output file from the ffindex output file, type:

```
$ ffindex2fasta DB_clu DB_clu.fasta
```

To run the more sensitive cascaded clustering and convert the result into FASTA format, type:

```
$ mmseqs_cluster DB DB_clu_s7 tmp --cascaded -s 7
$ ffindex2fasta DB_clu_s7 DB_clu_s7.fasta
```

4.4. System requirements

MMseqs runs under Linux only. Alignment and prefiltering modules are fully parallelized with SSE2 and OpenMP, i. e. MMseqs runs fastest on a computer with many (e.g. 16-32) cores. Besides, MMseqs needs much memory (you need 128G of memory in order to cluster the current UniProtKB version containing 54 M sequences). We offer an option for limiting the memory use at the cost of longer runtimes. The database is split into chunks and the program only holds one chunk in memory at any time. For clustering large databases containing tens of millions of sequences, you should provide enough free disc space (≈500 GB). In section 4.11, we will discuss the runtime, memory and disc space consumption of MMseqs and how to reduce resource requirements for large databases.

4.5. ffindex database format

All modules take ffindex databases as input and produce ffindex databases as output. ffindex was developed to avoid drastically slowing down the file system when millions of files need to be written and accessed. ffindex hides the files from the file system by storing them as unstructured data records in a single *data file*. In addition to this data file, an ffindex database includes a second file: This *index file* stores for each unique accession code the start position in bytes of the data record in the ffindex data file. When transforming a FASTA file with multiple sequences into an ffindex database, the accession code is the ID of the sequence parsed from the header. If no ID can be identified, the accession code is the whole header without the > character before the first blank space.

The binaries `fasta2ffindex` and `ffindex2fasta` located in `mmseqs/bin` do the format conversion from and to the ffindex database format. `fasta2ffindex` generates a ffindex database from a FASTA sequence database. `ffindex2fasta` converts an ffindex database to a FASTA formatted text file: the headers are ffindex accession codes preceded by >, with the corresponding dataset from the ffindex data file following.

However, for a fast access to the particular datasets in very large databases it is advisable

to use the ffindex database directly without converting. We offer the binary `ffindex_get` for direct access to the datasets stored in an ffindex database.

4.6. Overview of folders in MMseqs

- `bin`: MMseqs binaries, `fasta2ffindex` and `ffindex2fasta` binaries
- `data`: BLOSUM matrices and test data
- `lib`: ffindex sources and binaries
- `src`: MMseqs sources and the Makefile

4.7. Overview of MMseqs commands

MMseqs contains six binaries. Three commands execute workflows that combine MMseqs core modules. The other three commands execute the single modules which are used by the workflows and should be used by more advanced users.

Workflows:

- `mmseqs_search:` Compares all sequences in the query database with all sequences in the target database.
- `mmseqs_cluster:` Clusters the sequences in the input database by sequence similarity.
- `mmseqs_update:` Given an the existing clustering of a sequence database and a new version of the sequence database with some new sequences being added and others having been deleted, MMseqs incrementally updates the clustering.

Single modules:

- `mmseqs_pref:` Computes k-mer similarity scores between all sequences in the query database and all sequences in the target database.
- `mmseqs_aln:` Computes Smith-Waterman alignment scores between all sequences in the query database and the sequences of the target database whose prefiltering scores computed by `mmseqs_pref` pass a minimum threshold.
- `mmseqs_clu:` Computes a similarity clustering of a sequence database based on Smith Waterman alignment scores of the sequence pairs computed by `mmseqs_aln`.

4.8. Description of workflows

4.8.1. Batch sequence searching using `mmseqs_search`

For searching a database, you need your query and target database in ffindex format and an empty directory for MMseqs temporary files. Then, you can run the search by typing

```
$ mmseqs_search queryDB targetDB outDB tmp
```

To get more sensitive results, increase the search sensitivity (`-s` option):

```
$ mmseqs_search queryDB targetDB outDB tmp -s 7
```

The default sensitivity is 4, sensitivity can be set in the range $[1 : 9]$.

This workflow combines the prefiltering and alignment modules into a fast and sensitive batch protein sequence search that compares all sequences in the query database with all sequences in the target database. Query and target databases may be identical. The program outputs for each query sequence all database sequences satisfying the search criteria such as sensitivity of the search.

The underlying algorithm is explained in more detail in section 4.9.1, and the full parameter list can be found in section 4.12.1.

4.8.2. Clustering databases using `mmseqs_cluster`

For clustering a database, your need your sequence database in ffindex format and an empty directory for MMseqs temporary files. Then, you can run the clustering with

```
$ mmseqs_cluster inDB outDB tmp
```

and cascaded clustering with

```
$ mmseqs_cluster inDB outDB tmp --cascaded
```

For more sensitive clustering, adjust the sensitivity (`-s` option):

```
$ mmseqs_cluster inDB outDB tmp --cascaded -s 7
```

The clustering workflow combines the prefiltering, alignment and clustering modules into a simple clustering or cascaded clustering of a database. There are two possibilities to run the clustering:

- Simple clustering runs the prefiltering, alignment and clustering modules with predefined parameters.

- Cascaded clustering clusters the sequence database using prefiltering, alignment and clustering modules incrementally in three steps. In the first step, the prefiltering runs with a low sensitivity of 1 and a very high results significance threshold in order to accelerate the calculation and search only for hits with a high sequence identity. Then alignments are calculated and the database is clustered. The second step takes the representative sequences of the first clustering step and repeats the prefiltering, alignment and clustering steps. This time, the prefiltering is run with a higher sensitivity

and a lower result significance threshold for catching sequence pairs with lower sequence identity. In the third step, the whole process is repeated again with the target sensitivity defined by the `-s` parameter. Eventually, the clustering results are merged and the resulting clustering is written to the output ffindex database.

Cascaded clustering yields more sensitive results than simple clustering. Also, it allows very large cluster sizes in the end clustering resulting from cluster merging (note that cluster size can grow exponentially in the cascaded clustering workflow), which is not possible with the simple clustering workflow because of the limited maximum number of sequences passing the prefiltering and the alignment. Therefore, we strongly recommend to use cascaded clustering especially to cluster larger databases and to obtain maximum sensitivity.

The underlying algorithm is explained in more detail in section 4.9.1, and the full parameter list can be found in section 4.12.2.

4.8.3. Updating a database clustering using `mmseqs_update`

To run the updating, you need the old and the new version of your sequence database in ffindex format, the clustering of the old database version and a directory for the temporary files:

```
$ mmseqs_update oldDB newDB oldDB_clustering outDB tmp
```

This workflow efficiently updates the clustering of a database by adding new and removing outdated sequences. It takes as input the older sequence database, the results obtained by this older database clustering, and the newer version of the sequence database. Then it adds the new sequences to the clustering and removes the sequences that were removed from the newer database. Sequences which are not similar enough to any existing cluster will be founders of new clusters.

4.9. Description of core modules

For advanced users, it is possible to run core modules for maximum flexibility. Especially for the sequence search it can be useful to adjust the prefiltering and alignment parameters according to the needs of the user. The detailed parameter lists for the modules is provided in section 4.12.

MMseqs contains three core modules: prefiltering, alignment and clustering.

4.9.1. Computation of prefiltering scores using `mmseqs_pref`

The prefiltering module calculates the sum of scores of similar k-mers between all query sequences and all database sequences and outputs the most similar sequence pairs.

If you want to *cluster* a database, or do an all-against-all search, you only have one input database. In this case, the prefiltering does an all-against-all search with the following

program call:

```
$ mmseqs_pref inputDB inputDB resultDB_pref
```

`inputDB` is the base name of the ffindex databases you produced from your FASTA sequence databases, the prefiltering results are stored in the ffindex database files `resultDB_pref`, `resultDB_pref.index`.

If you want to do a *sequence search*, you have two input databases: query database and target database, that also can be identical. In this case, the prefiltering program call is:

```
$ mmseqs_pref queryDB targetDB resultDB_pref
```

First, the database sequences are indexed in an index table to provide a fast access to the k-mers of the database sequences. The index table has an array with a pointer for each possible k-mer to an index list storing the IDs of the database sequences containing this k-mer. After the index table generation, the algorithm processes each query sequence from left to right and generates a list of similar k-mers for the k-mer at the current query sequence position. For each k-mer in the list, the k-mer similarity score is added to the overall prefiltering score for each database sequence containing this k-mer, retrieved using the index table. After processing the whole query sequence, database sequences with significant prefiltering scores are extracted and written to the prefiltering result database.

Since different queries yield different score distribution in the database, a rigid prefiltering score threshold does not work well. The statistical significance of a prefiltering score for a given query sequence is described by the Z-score. The Z-score for a prefiltering score of a query sequence and a database sequence is calculated based on the score distribution for the query in the database. For each query and database sequence pair, the prefiltering score S_{qt} is normalized by subtracting the background score S_0 expected by chance and dividing by the standard deviation of the score σ_S, resulting in a normalized Z-score Z_{qt}:

$$Z_{qt} = \frac{S_{qt} - S_0}{\sigma_S}$$

Instead of setting a prefiltering score threshold, we set a rigid Z-score (i. e. result significance) threshold. Only results with a sufficient Z-score are written to the output.

The sensitivity of the prefiltering can be set using the `-s` option. Internally, `-s` sets the average length of the lists of similar k-mers per query sequence position and the Z-score threshold.

- *Similar k-mers list length:* Low sensitivity yields short similar k-mer lists. Therefore, the speed of the prefiltering increases, since only short k-mer lists have to be generated and less lookups in the index table are necessary. However, the sensitivity of the search decreases, since only very similar k-mers are generated and therefore, the prefiltering can not identify sequence pairs with low sequence identity.

- *Z-score threshold:* Z-score of a prefiltering result describes its statistical significance.

Lower sensitivity yields a higher Z-score threshold, i. e. only the most significant results are displayed.

Furthermore, there is a possibility to use different lengths of the k-mers used in the prefiltering. Longer k-mers are more sensitive, since they cause less chance matches. On the other hand, for a fixed run time, longer k-mers only pay off for larger databases. The reason is the different relation between the time for the k-mer list generation and database matching for different database sizes. Longer k-mers need more time for the k-mer list generation, but less time for database matching. Therefore, the database matching should take most of the computation time, which is only the case for large databases. For a fixed run time, the default value $k = 6$ is the best for databases containing a few million sequences. For very large databases containing about 100 million sequences, $k = 7$ should be a better choice theoretically, though the real life performance of 7-mers on large databases is not tested yet. For databases containing only hundreds of thousands of sequences, $k = 5$ should be sufficient.

4.9.2. Local alignment of prefiltering sequences using `mmseqs_align`

In the alignment module, you can also specify either identical or different query and target databases. If you want to do a clustering in the next step, query and target database need to be identical.

```
$ mmseqs_align inputDB inputDB resultDB_pref resultDB_aln
```

Alignment results are stored in the ffindex files `resultDB_aln`, `resultDB_aln.index`.

Program call in case you want to do the sequence search and have different query and target databases:

```
$ mmseqs_align queryDB targetDB resultDB_pref resultDB_aln
```

This module implements an SSE2-accelerated Smith-Waterman-alignment (Farrar, 2007) of all sequences that pass a cut-off for the prefiltering score in the first module. It processes each sequence pair from the prefiltering results and aligns them in parallel, calculating one alignment per core at a single point of time. Additionally, the alignment calculation is vectorized using SIMD (single instruction multiple data) instructions. Eventually, the alignment module calculates alignment statistics such as sequence identity, alignment coverage and e-value of the alignment.

4.9.3. Clustering sequence database using `mmseqs_clu`

For the clustering, you need the input sequence database and the alignment results for the database:

```
$ mmseqs_clu inputDB resultsDB_aln resultsDB_clu
```

Clustering results are stored in the ffindex database files `resultsDB_clu`, `resultsDB_clu.index`.

The clustering module offers the possibility to run two different clustering algorithms. A greedy set cover algorithm is the default. It tries to cover the database by as few clusters as possible. At each step, it forms a cluster containing the representative sequence with the most alignments above the special or default thresholds with other sequences of the database and these matched sequences. Then, the sequences contained in the cluster are removed and the next representative sequence is chosen.

The other clustering algorithm is a greedy clustering algorithm, as used in CD-HIT. It sorts sequences by length and in each step forms a cluster containing the longest sequence and sequences that it matches. Then, these sequences are removed and the next cluster is chosen from the remaining sequences.

Note that we *always* recommend to use the cascaded clustering workflow instead of the clustering module for larger databases, since the maximum cluster size is limited to a quite low value otherwise (between 50 and 300 for large databases containing millions of sequences, depending on the database size). The reasons are the limited result list length in the prefiltering and alignment modules (the maximum list length determines the maximum cluster size in the simple clustering workflow) and the high memory consumption of the clustering for large databases with many alignment results per query.

4.10. Output file formats

Results of MMseqs commands are stored in ffindex databases. All records within those ffindex databases are in plain text format.

4.10.1. Prefiltering

The ffindex accession code is the UniProtKB ID (or other ID depending on the database format) of the query. A line in the prefiltering result database record (= one match) has the following format:

```
targetId Z-score prefilteringScore
```

where `targetId` is the database ID of the matched sequence, `Z-score` is the statistical significance score of the match and `prefilteringScore` is the raw score of the match (the sum of the scores of similar k-mers of the query and target sequence) in half bits. Example of a prefiltering result for the SwissProt sequence Q54G30 (excerpt):

```
Q54G30 1177.95 55735
Q869W0 159.179 5274
Q86IM3 99.2044 1823
Q54E43 85.8743 3224
```

The first match is the identity Q54G30 having a very high prefiltering score of 55735 and the Z-score of 1177.95.

4.10.2. Alignment

The ffindex accession code is the UniProtKB ID (or other ID depending on the database format) of the query. One line of the alignment results record has the following format:

`targetId alnScore queryCov targetCov seqId evalue`

where `targetId` is the database ID of the matched sequence, `alnScore` is the raw score of the alignment in half bits, `queryCov` is the alignment coverage of the query in the range $[0 : 1]$, `targetCov` is the alignment coverage of the target database sequence in the range $[0 : 1]$, `seqId` is the sequence identity and `evalue` is the e-value of the match. Example of an alignment result for the SwissProt sequence A0PUH6 (excerpt):

```
A0PUH6 1305 1.000 1.000 1.000 1.507e-186
Q6NFN4 824 0.956 0.974 0.649 3.682e-114
Q8DD39 256 0.900 0.909 0.335 1.136e-28
P52973 182 0.808 0.822 0.238 1.597e-17
```

The first line is the identity match. The last sequence P52973 has a Smith-Waterman alignment score 182, query sequence coverage 0.808, database sequence coverage 0.822, the alignment has the sequence identity 0.238 and the e-value 1.597e-17.

4.10.3. Clustering

Every cluster is stored once (i. e. one result database record per cluster). Each database record contains UniProtKB IDs (or other IDs depending on the database format) of the sequences assigned to this cluster, one ID per line. The ffindex accession code is the ID of the representative sequence of the cluster. An example of a cluster record with 4 cluster members:

```
Q9ZZZ1
Q96189
O03850
P03887
```

4.11. Optimizing sensitivity and consumption of resources

This section discusses how to keep the run time, the memory and disc space consumption of MMseqs at reasonable values, while obtaining results with the highest possible sensitivity. These considerations are relevant if the size of your database exceeds several millions of sequences and they are most relevant if the database size is in the order of tens of millions of sequences.

4.11.1. Prefiltering module

The prefiltering module can use a lot of resources regarding all the memory consumption, the runtime and the disc space, if the parameters are not set appropriately.

Memory consumption For maximum efficiency of the prefiltering, the entire database should be held in RAM memory. The major part of memory is required for the k-mer index table of the database. For a database containing N sequences with an average length L, the memory consumption of the index lists is $N \times L \times 4\text{B}$. Note that the memory consumption grows linearly with the size of the sequence database. In addition, the index table stores the pointer array and two auxiliary arrays with the memory consumption of $a^k \times 16\text{B}$, where a is the size of the amino acid alphabet (usually 21 including the unknown amino acid X) and k is the k-mer size. The overall memory consumption of the index table is

$$M \quad = \quad (4\,N\,L + 16\,a^k)\text{B}$$

Therefore, the UniProtKB database version of April 2014 containing 55 million sequences with an average length 350 needs about 71 GB of main memory.

To limit the memory use at the cost of longer runtimes, the option `--max-chunk-size` allows the user to split the database into chunks of the given maximum size.

Runtime The prefiltering module is the most time consuming step. To cluster the 55 million sequences of UniProtKB (04/2014), the MMseqs prefiltering module needs about 6 days when running on 32 cores and about 10 days when running on 16 cores of a modern computer.

Disc space The prefiltering results for very large databases can grow to considerable sizes (in the order of TB) of the disc space if very long result lists are allowed and a low Z-score threshold is set. As an example, an all-against-all prefiltering run on the UniProtKB with `--max-seqs` 300 yielded average prefiltering list length 150 and the output file size 146 GB.

Important options for tuning the memory, runtime and disc space usage

- The option `-s` controls the sensitivity in the MMseqs prefiltering module. The lower the sensitivity, the faster the prefiltering gets at the cost of search sensitivity. Default sensitivity is 4, increasing the sensitivity by one roughly doubles the runtime of the prefiltering. In order to cluster the UniProtKB down to ≈30% sequence identity, you should leave this parameter at the default value of 4. For clustering down to 90%, sensitivity 1 should be sufficient, although there are still no specific tests for the optimum parameters necessary for clustering down to a fixed sequence identity.
- The option `--max-seqs` controls the maximum number of prefiltering results per query sequence. For very large databases (tens of millions of sequences), it is a good advice to keep this number at reasonable values (i. e. the default value 300). For considerably larger values of `--max-seqs`, the size of the output can be in the range

of several TB of disc space for databases containing tens of millions of sequences. Changing `--max-seqs` option has no effect on the run time.

- The option `--z-score` describes the minimum significance of the results written to the output. Usually, this option is set automatically depending on the sensitivity. However, especially for the sequence search it can be desired to see also less significant results. Setting `--z-score` at lower values yields more results and therefore increases the size of the output written to disc. In addition, it slows down the program.

4.11.2. Alignment module

In the alignment module, generally only the runtime is a critical issue.

Memory consumption The major part of the memory is required for the three dynamic programming matrices, once per core. Since most sequences are quite short, the memory requirements of the alignment module for a typical database are in the order of a few GB.

Runtime It takes about 2-3 days to compute Smith-Waterman alignments for the UniProtKB sequence pairs which passed the prefiltering step (at default parameters for deep clustering down to ≈20 - 30% pairwise sequence identity).

If a huge amount of alignments has to be calculated, the run time of the alignment module can become a bottleneck. The run time of the alignment module depends essentially on two parameters:

- The option `--max-seqs` controls the maximum number of sequences aligned with a query sequence. By setting this parameter to a lower value, you accelerate the program, but you may also lose some meaningful results. Since the prefiltering results are always ordered by their significance, the most significant prefiltering results are always aligned first in the alignment module.

- The option `--max-rejected` defines the maximum number of rejected sequences for a query until the calculation of alignments stops. A reject is an alignment whose statistics don't satisfy the search criteria such as coverage threshold, e-value threshold etc. Per default, `--max-rejected` is set to `INT_MAX`, i. e. all alignments until `--max-seqs` alignments are calculated.

Disc space Since the alignment module takes the results of the prefiltering module as input, the size of the prefiltering module output is the point of reference. If alignments are calculated and written for all the prefiltering results, the disc space consumption is 1.75 times higher than the prefiltering output size.

4.11.3. Clustering module

In the clustering module, only the memory consumption is a critical issue.

Memory consumption The clustering module can need large amounts of memory. The memory consumption for a database containing N sequences and an average of r alignment results per sequence can be estimated as

$$M \quad = \quad 40 \times N \times r \, \mathrm{B}$$

To prevent excessive memory usage for the clustering of large databases, you should use cascaded clustering (`--cascaded` option) which accumulates sequences per cluster incrementally, therefore avoiding excessive memory use.

If you run the clustering module separately, you can tune

- `--max-seqs` parameter which controls the maximum number of alignment results per query considered (i. e. the number of edges per node in the graph). Lower value causes lower memory usage and faster run times.
- Alternatively, `-s` parameter can be set to a higher value in order to cluster the database down to higher sequence identities. Only the alignment results above the sequence identity threshold are imported and it results in lower memory usage.

Runtime Clustering is the fastest step. It needs about 2 hours for the clustering of the whole UniProtKB.

Disc space Since only one record is written per cluster, the memory usage is a small fraction of the memory usage in the prefiltering and alignment modules.

4.11.4. Workflows

The search and clustering workflows offer the possibility to set the sensitivity option `-s` and the maximum sequences per query option `--max-seqs`. `--max-rejected` option is set to `INT_MAX` per default. Cascaded clustering sets all the options controlling the size of the output, speed and memory consumption, internally adjusting parameters in each cascaded clustering step.

4.12. Detailed parameter list

4.12.1. Search workflow

Compares all sequences in the query database with all sequences in the target database.

Usage:

```
mmseqs_search <queryDB> <targetDB> <outDB> <tmpDir> [opts]
```

Options:

`-s [float] Target sensitivity in the range [1:9] (default=4).`

Adjusts the sensitivity of the prefiltering and influences the prefiltering run time. For detailed explanation see section 4.9.1.

`--z-score [float] Z-score threshold (default: 50.0)`

Prefiltering Z-score cutoff. A lower z-score cutoff yields more results, since also less significant results are written to the output. For detailed explanation see section 4.9.1.

`--max-seqs Maximum result sequences per query (default=300)`

Maximum number of sequences passing the prefiltering and alignment per query. If the prefiltering result list exceeds the `--max-seqs` value, only the sequences with the best Z-score pass the prefiltering and are aligned in the alignment step.

`--max-seq-len [int] Maximum sequence length (default=50000).`

The length of the longest sequence in the input database.

`--sub-mat [file] Amino acid substitution matrix file (default: BLOSUM62).`

Substitution matrices for different sequence diversities in the required format can be found in the MMseqs `data` folder.

4.12.2. Clustering workflow

Calculates the clustering of the sequences in the input database.

Usage:

```
mmseqs_cluster <sequenceDB> <outDB> <tmpDir> [opts]
```

Options:

`--cascaded Start the cascaded instead of simple clustering workflow.`

The database is clustered incrementally in three steps and improves the sensitivity of the clustering greatly compared to the general workflow. For detailed explanation, see the section 4.8.2.

`-s [float] Target sensitivity in the range [2:9] (default=4).`

Adjusts the sensitivity of the prefiltering and influences the prefiltering run time. For detailed explanation see section 4.9.1.

`--max-seqs Maximum result sequences per query (default=300).`

Maximum number of sequences passing the prefiltering and alignment per query. If the prefiltering result list exceeds the `--max-seqs` value, only the sequences with the best Z-score pass the prefiltering and are aligned in the alignment step.

`--max-seq-len [int] Maximum sequence length (default=50000).`

The length of the longest sequence in the database.

`--sub-mat [file] Amino acid substitution matrix file.`

Substitution matrices for different sequence diversities in the required format can be found in the MMseqs `data` folder.

4.12.3. Updating workflow

Updates the existing clustering of the previous database version with new sequences from the current version of the same database.

Usage:

```
mmseqs_update <oldDB> <newDB> <oldDB_clustering> <outDB> <tmpDir> [opts]
```

Options:

```
--sub-mat [file] Amino acid substitution matrix file.
```

Substitution matrices for different sequence diversities in the required format can be found in the MMseqs `data` folder.

```
--max-seq-len [int] Maximum sequence length (default=50000).
```

The length of the longest sequence in the database.

4.12.4. Prefiltering

Calculates k-mer similarity scores between all sequences in the query database and all sequences in the target database.

Usage:

```
mmseqs_pref <queryDB> <targetDB> <outDB> [opts]
```

Options:

```
-s [float] Sensitivity in the range [1:9] (default=4).
```

Adjusts the sensitivity of the prefiltering and influences the prefiltering run time. For detailed explanation see section 4.9.1.

```
-k [int] k-mer size in the range [4:7] (default=6).
```

The size of k-mers used in the prefiltering. For guidelines for choosing a different k as the default, see section 4.9.1.

```
-cpu [int] Number of cores used for the computation (default=all cores).
--alph-size [int] Amino acid alphabet size (default=21).
```

Amino acid alphabet size, default = 21 (full amino acid alphabet). For using a reduced amino acid alphabet, choose a lower value. Reduced amino acid alphabets reduce the memory usage, but also the sensitivity.

```
--z-score [float] Z-score threshold (default:  50.0).
```

Prefiltering Z-score cutoff. A lower z-score cutoff yields more results, since also less significant results are written to the output. For detailed explanation see section 4.9.1.

```
--max-seq-len [int] Maximum sequence length (default=50000).
```

The length of the longest sequence in the database.

```
--nucl Nucleotide sequences input.
--max-seqs [int] Maximum result sequences per query (default=300).
```

Maximum number of sequences passing the prefiltering per query. If the prefiltering result list exceeds the `--max-seqs` value, only the sequences with the best Z-score pass the prefiltering.

```
--no-comp-bias-corr Switch off local amino acid composition bias correction.
```

Compositional bias correction assigns lower scores to amino acid matches of the amino acids that are frequent in their neighborhood in the query sequence.

`--max-chunk-size [int] Splits target databases in chunks when the database size exceeds the given size. (For memory saving only)`

Maximum number of sequences stored in the index table at some point of time, default = `INT_MAX`. Restraining the number of sequences stored reduces the memory usage, but slows down the calculation.

`--skip [int] Number of skipped k-mers during the index table generation.`

Number of k-mers in the database sequences skipped during the index table generation. Per default, each k-mer of the database is indexed. With skip = 2, 2 k-mers are skipped and only each third k-mer is indexed. This speeds up the search and reduces the memory usage at the cost of lower search sensitivity.

`--sub-mat [file] Amino acid substitution matrix file.`

Substitution matrices for different sequence diversities in the required format can be found in the MMseqs `data` folder.

`-v [int] Verbosity level: 0=NOTHING, 1=ERROR, 2=WARNING, 3=INFO (default=3).`

Verbosity level in the range $[0 : 3]$. With verbosity 0, there is no terminal output.

4.12.5. Alignment

Calculates Smith-Waterman alignment scores between all sequences in the query database and the sequences of the target database which passed the prefiltering.

Usage:

`mmseqs_pref <queryDB> <targetDB> <prefResultsDB> <outDB> [opts]`

Options:

`-e [float] Maximum e-value (default=0.01).`

E-value of the local alignment is calculated using Karlin-Altschul statistics.

`-c [float] Minimum alignment coverage (default=0.8).`

Minimum alignment coverage of both query and database sequence, default = 0.8. With the value of 0.0, the alignments are assessed using only the e-value criterion.

`-cpu [int] Number of cores used for the computation (default=all cores).`

`--max-seq-len [int] Maximum sequence length (default=50000).`

The length of the longest sequence in the database.

`--max-seqs [int] Maximum alignment results per query sequence (default=300).`

Maximum number of sequences passing the alignment per query. Sequences are read in the order of the prefiltering lists. The reading for a query is stopped if the number of sequences for a query sequence exceeds the `--max-seqs` value.

`--max-rejected [int] Maximum rejected alignments before alignment calculation for a query is aborted. (default=INT_MAX)`

Maximum number of rejected alignments for a query until the alignment calculation is stopped. A rejected alignment is an alignment that does not satisfy the e-value and

alignment coverage thresholds. Default = `INT_MAX` (i. e., all alignments are calculated).

`--nucleotides Nucleotide sequences input.`

`--sub-mat [file] Amino acid substitution matrix file.`

Substitution matrices for different sequence diversities in the required format can be found in the MMseqs `data` folder.

`-v [int] Verbosity level: 0=NOTHING, 1=ERROR, 2=WARNING, 3=INFO (default=3).`

Verbosity level in the range [0 : 3]. With verbosity 0, there is no terminal output.

4.12.6. Clustering

Calculates a clustering of a sequence database based on Smith Waterman alignment scores of the sequence pairs.

Usage:

`mmseqs_clu <sequenceDB> <alnResultsDB> <outDB> [opts]`

Options:

`-g greedy clustering by sequence length (default: set cover clustering algorithm).`

Use a greedy clustering algorithm instead of the set cover algorithm. For the description of the two algorithms, see section 4.9.3.

`-s [float] Minimum sequence identity of sequences in a cluster (default = 0.0)`

Minimum sequence identity of the cluster members and the representative sequence. Per default, the sequence identity criterion is switched off.

`--max-seqs [int] Maximum result sequences per query (default=100)`

Maximum alignment results read per query. This is at the same time the maximum possible number of sequences in the cluster.

`-v [int] Verbosity level: 0=NOTHING, 1=ERROR, 2=WARNING, 3=INFO (default=3).`

Verbosity level in the range [0 : 3]. With verbosity 0, there is no terminal output.

4.13. License terms

The software is made available under the terms of the GNU General Public License v3.0. Its contributors assume no responsibility for errors or omissions in the software.

5. ROC5 vs. ROC

We usually use ROC5 plots to assess the performance of the sequence search. To build a ROC5 plot, ROC5 score of each query is calculated, i. e. the fraction of true positives discovered before the fifth false positive occurs when the results for each query are ordered by the score. Then, all these ROC5 values are sorted and a cumulative plot is generated.

In a ROC plot, all search results are ordered by the score and each result is labeled with false positive or true positive. Then, a plot is generated by plotting the cumulative number of true positives versus the number of false positives for each result, starting with the highest score and continuing with decreasing scores.

We prefer to use ROC5 plots, since they produce a more balanced picture of tool performance. In a ROC5 plot, each query contributes equally to the plot. In a ROC plot, few queries producing high scores with the database can dominate the picture. In addition, queries with a large number of results contribute much more to the plot than queries having only a few search results. This can be the case, for instance, if someone does a sequence search against SCOP database, which contains some very large and some small protein families. A sequence that belongs to a large protein family will get more results and therefore contribute much more to ta ROC plot, than a sequence stemming from a protein family with only a few members. In the ROC5 plot, both sequences will contribute equally.

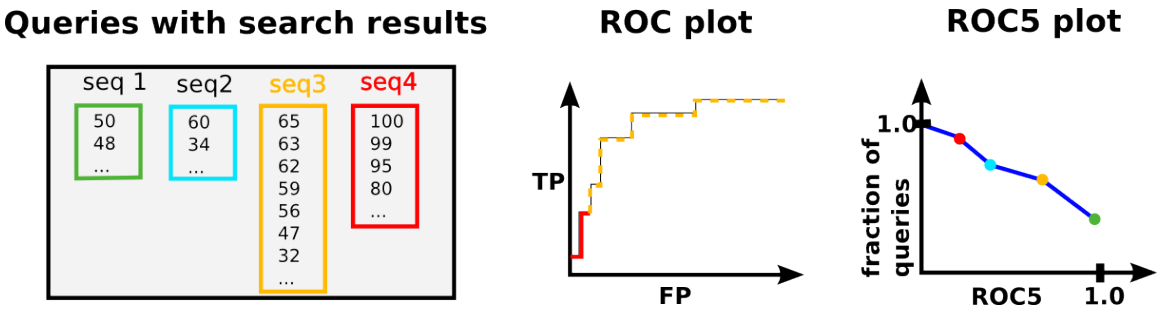


Figure 5.1.: **ROC plot bias and the comparison to ROC5 plot.** The bias of the ROC plot is shown on an imaginary example. A database of queries is shown where sequence 3 (yellow) produces a vast bulk of search results (e. g. because it is member of a very large protein family) and sequence 4 (red) produces very high scoring results (e. g. it is a very long sequence and the score is not corrected for the query length). In the ROC plot, sequence 3 contributes disproportionately many values to the beginning of the ROC curve, while sequence 4 contributes disproportionately many values to the whole curve. In the ROC5 plot, each sequence contributes only one ROC5 value to the curve. The ROC5 value depends on the proportion of the true positives before the 5th false positive in the result list.

Figure A.1 demonstrates an example where the ROC5 plot shows more balanced picture of the search results for some query sequences.

6. Results

6.1. Prefiltering quality benchmark

We benchmarked different MMseqs prefiltering settings in order to determine the most effective prefiltering setting. We assessed the quality of the prefiltering results by measuring its ability to discover sequences producing significant Smith-Waterman alignments with the query and visualized the results in a ROC5 and a ROC plot. The results are shown in Figure 6.1a.

For the benchmark, we draw 13 000 query sequences and 300 000 database sequences from UniRef100 (a non-redundant set of sequences from UniProtKB) (Suzek et al., 2007). For each query, we calculated Smith-Waterman alignments with all database sequences. Then, we ran the prefiltering module with different k-mer sizes and different sensitivity settings and compared the results to the Smith-Waterman results. A true positive sequence pair has a Smith-Waterman alignment e-value < 0.01. All sequence pairs with the sequence identity < 0.3 and $0.01 < \mathrm{e-value} < 10.0$ were removed from the benchmark, because they were considered as either too hard to detect or as not clearly classified as homologs.

Then the prefiltering results for each query are ordered by their prefiltering score and a ROC and a ROC 5 plot is produced. ROC5 plot is generally more reliable than the ROC plot because each query contributes equally to the ROC5 plot, whereas in a ROC plot some queries with high scores and many matches in the database can dominate the overall picture (see also chapter 5 for the ROC5 plot discussion).

ROC and ROC5 curves are generated for $k = [4,7]$ and MMseqs sensitivity 4. Additionally, for $k = 6$ and $k = 7$, sensitivities 7 and 9 are tested. Other parameters used are `--max-seqs 1000` and `--z-score-thr 5.0` in order to increase the number of results output for each query and reliably measure the ROC5 score.

When setting sensitivity to 4 (solid lines), 6-mers are performing best. With increased sensitivity, the performance of 7-mers improves, while the performance of 6-mers stagnates. The reason is that 7-mers need longer k-mer similarity lists to yield better results than 6-mers. So, at lower sensitivity, lists of similar 7-mers are too short. In contrast, increasing the similar 6-mer list length does not bring any performance improvement at any point, since the 6-mers get too dissimilar to the query sequence k-mer and rather blur than improve the prefiltering results for a query. In the ROC5 curve in the Figure 6.1a, one can see that 6-mers performance with sensitivity 9 even drops a bit compared to $s = 7$.

In addition, in section 3.2.8, we have hypothesized that the minimum database size for

7-mers where matching of the index table and not the k-mer list generation predominates the running time is 12 M sequences. The database we are using here is much smaller, so it can be expected that 7-mers would show a much better performance if we increase the size of the database by 15-fold.

The default sensitivity in MMseqs prefiltering is $s = 4$, since it offers the best trade-off between the result quality and the runtime. We chose the default length 6 for the k-mers based on the benchmark results.

6.2. Protein sequence searching

6.2.1. Benchmarked methods and parameters

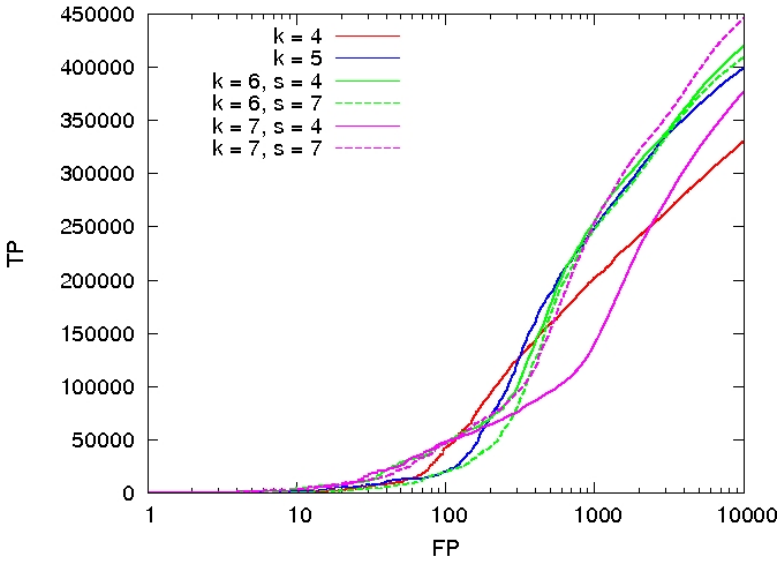
Our goal was to assess the quality and the speed of protein search with different protein search tools. We benchmarked MMseqs together with other popular protein search methods: Smith-Waterman alignments, BLAST, UBLAST and RAPsearch. We assessed the quality of protein search results on SCOP25-based protein sequence set, checking how well each tool discovers homologous sequences for the query. The results are shown in a ROC and a ROC5 plot. For the speed assessment, we searched a set of protein sequences against the whole UniProtKB.

We set parameters for each tool such that long result lists are produced for each query. This is necessary for the calculation of ROC5 values, since we are looking for the number of true positives before the fifth false positive occurs, so the result list should be long enough to contain at least five false positives.

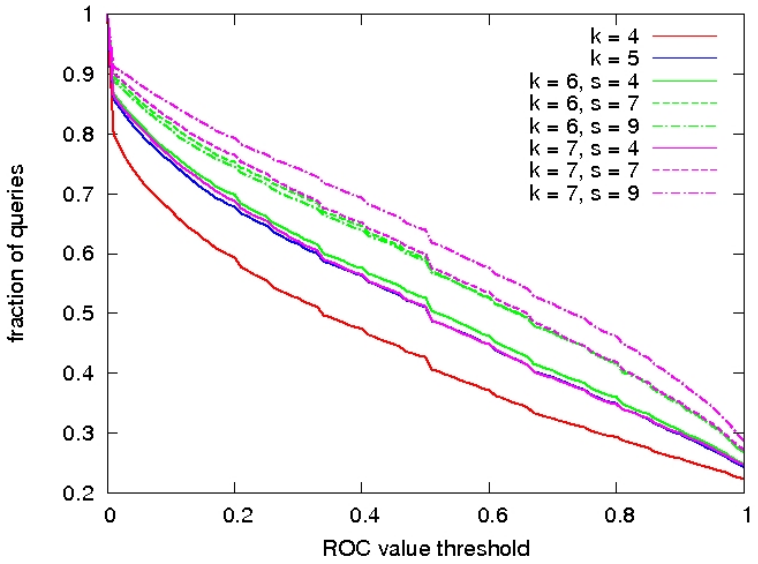
All runs were made on a 16 cores (processors: Intel Xeon E5-2680, 2.70GHz) computer with a 128 GB RAM.

MMseqs We used only the prefiltering module and the alignment module of MMseqs for the protein search. We tested two different sensitivities in the prefiltering module, $s = 4$ (default setting) and $s = 7$. Besides that, we set the maximum prefiltering list length to 1000 using `--max-seqs 1000`, and the Z-score threshold to 10.0 using `--z-score-thr 10.0` in order to increase the length of resulted lists of each query. The alignment module was run with the maximum e-value threshold 10.0 using `-e 10.0`. The alignment coverage was switched off using `-c 0.0`. Both modules use all 16 cores of the machine per default.

Smith-Waterman alignments with swipe We used a SSE2-, multi-core-parallelized Smith-Waterman alignment calculation with swipe. In order to get many database matches for a query, we set the e-value to 100 using `-e 100.0` and the number of sequence descriptions and sequence alignments to 10 000 using `-v 10000 -b 10000`. Additionally, swipe was using all the 16 cores of the machine with `-a 16`.



(a) ROC plot for the performance of the prefiltering module with different settings.



(b) ROC5 plot for the performance of the prefiltering module with different settings.

Figure 6.1.: **Performance of MMseqs prefiltering module.** Performance of 4-mers, 5-mers, 6-mers and 7-mers is compared, using Smith-Waterman scores as the golden standard. k specifies the k-mer size, s the MMseqs sensitivity value. a) ROC plot for the different prefiltering settings. 4-mers and 5-mers are tested with the default sensitivity 4, 6-mers and 7-mers are additionally tested with sensitivity 7 (dashed lines). b) For each query, a ROC5 value is calculated - i.e., the fraction of true positives found for the query until the fifth false positive occurs. Then, the fraction of queries having a ROC5 value smaller or equal to a ROC5 threshold is plotted against this ROC5 threshold for each ROC5 in the range [0,1]. Solid line shows the results for sensitivity 4, dashed line for sensitivity 7, dashed and dotted line for sensitivity 9.

BLAST We ran BLAST using `-e 10.0` and `-v 10000 -b 10000`, in order to increase the number of results, and `-a 16` to parallelize the calculation.

UBLAST We ran UBLAST with `-evalue 10.0` option. Therefore, UBLAST outputs all significant alignments regardless of the sequence identity and alignment coverage. Ublast uses all available cores per default.

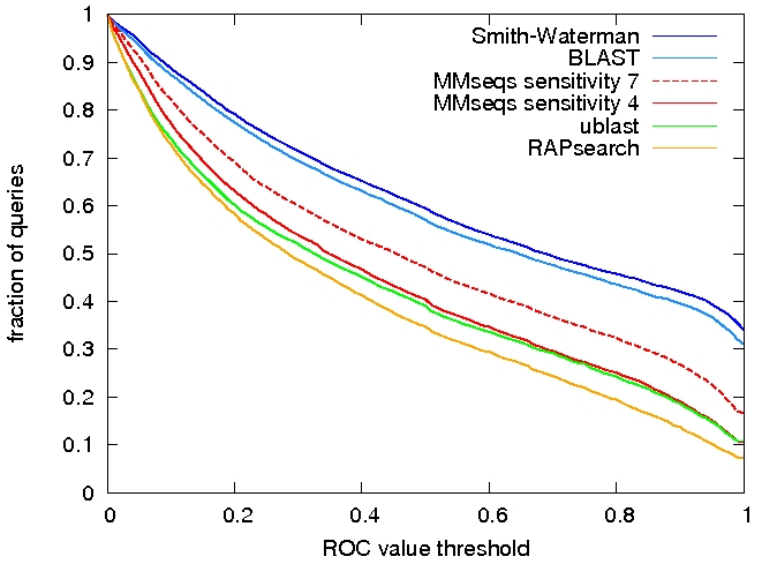
RAPsearch We ran RAPsearch with `-z 16` option to parallelize the calculation, and with `-e 10.0` and `-v 10000 -b 10000` options.

6.2.2. Protein sequence searching results

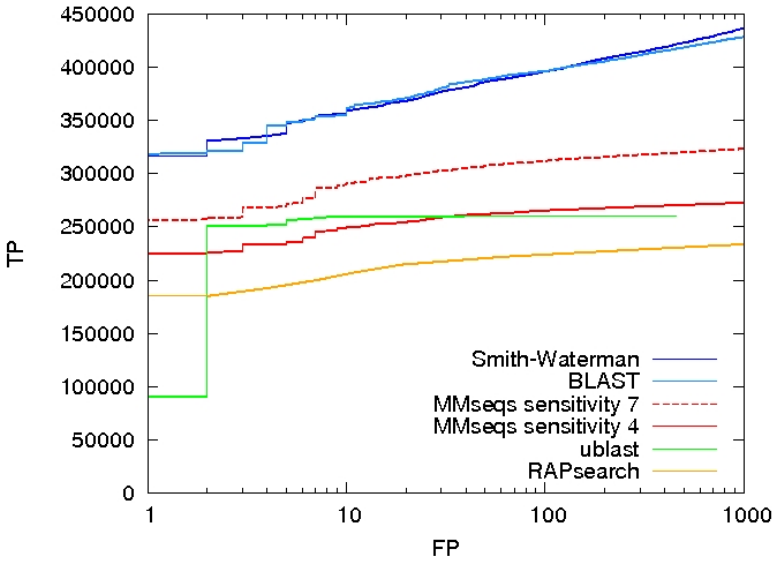
We used a dataset based on the SCOP database (Lo Conte et al., 2000) in order to assess the protein search ability of the tools. For generating the database, we ran one iteration of HHblits with the sequences from SCOP25 (Brenner et al., 2000), i. e. the SCOP database filtered down to maximum 25% pairwise sequence identity, against the whole UniProtKB. The resulting alignments were filtered such that the alignment with the database sequence covers a minimum of 80% of the query SCOP25 sequence, with minimum 30% and maximum 80% sequence identity. Further, the alignments were filtered to keep the most diverse set of sequences for each query, at least 50 sequences for each query. Eventually, all sequences containing inserted domains and all "broken" sequences containing only parts of domains were removed from the database using very sensitive HHblits searches. The resulting database contained 283 406 sequences. The query set was SCOP25 (7 616 sequences).

The performance of single methods is shown in Figure 6.2. For each query from SCOP25, the sequences from UniProtKB homologous to that sequence are labeled with the superfamily of the query. Then, ROC5 curve is generated for the results from each protein search tool. A true positive is a match of the query with a the sequence in the database, that is labeled with the same superfamily as the query sequence. A false positive is a database sequence labeled with a different fold than the query. All search results with a different superfamily but the same fold are ignored.

Smith-Waterman alignments yielded the best performance, with BLAST performance being slightly inferior. MMseqs with default sensitivity 4 performs slightly better than UBLAST, and it improves considerably when setting the sensitivity to 7. Noteworthy are the high-scoring false positives UBLAST produces that can be seen in the ROC plot. The reason is a bug in UBLAST which leads to the wrong labeling of the sequences in some cases (some headers are changed). The UBLAST ROC curve is shorter because of the overall lower number of discovered hits.



(a) ROC5 plot for the protein sequence search performance of different methods.



(b) ROC plot for the protein sequence search performance of different methods.

Figure 6.2.: **Protein search results of the different tools.** Protein search ability of MMseqs, Smith-Waterman alignments, BLAST, UBLAST and RAPsearch. 7 616 queries from SCOP25 were searched against the database containing 283 406 sequences homologous to some sequence from SCOP25. A true positive is a SCOP sequence and the UniProtKB sequence homologous to a SCOP25 sequence from the same superfamily as the query. A false positive is a pair of sequences with different folds. All pairs of sequences with different families but the same fold are ignored. The reason for the jump in the UBLAST ROC curve is a bug in UBLAST causing sometimes wrong assignment of the sequence headers to the sequences. The UBLAST ROC curve is shorter because of the overall lower number of discovered hits.

6.2.3. Benchmark for short-read searching sensitivity

In addition to the previous sequence search benchmark, we tested MMseqs and other tools on a short read dataset. The ability to map short reads to the longer database sequences correctly is an important application, i. e. in analysis of metagenomics data. Short reads where no reference genome exist can be mapped on known proteins contained in the UniProtKB database and annotated based on the matches. Although many tools exist which can map short reads with very few substitutions or deletions on a reference genome very fast (Langmead et al., 2009; Li and Durbin, 2009; Li et al., 2008; Trapnell and Salzberg, 2009), there is a need for tools which search for more diverged homologous sequences in a large protein sequence database. In the past, BLAST was widely used for homology searches, but recently, BLAST has become too slow for searches in the constantly growing databases and analyzing huge NGS data amounts. Therefore, we wanted to test MMseqs' performance in search of short reads for homologous protein sequences with low sequence identity.

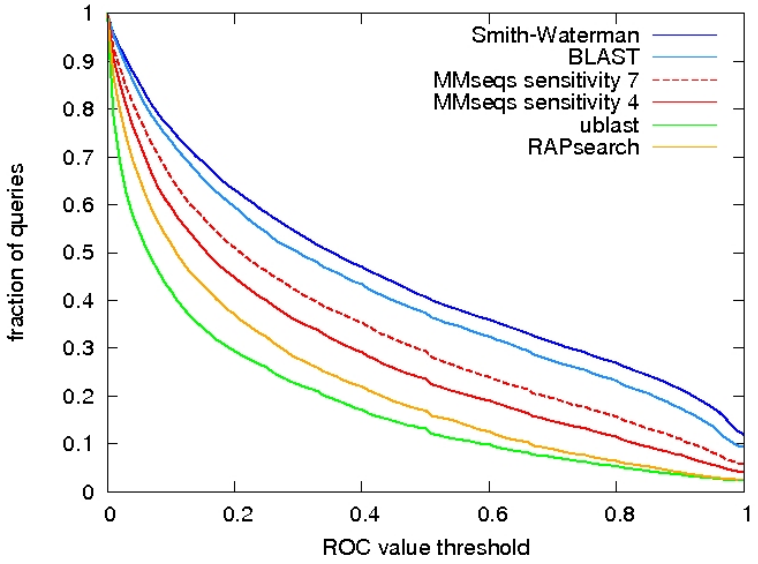
We generated two simulated short read datasets by randomly cutting sequence pieces from the query sequences from the previous benchmark (SCOP25 protein domains). The length of the sequence pieces was 50 for the first and 100 for the second dataset. Then, we did a sequence search for these two datasets against the same target database as in the previous benchmark. We tested the performance of the same methods as in the previous search benchmark.

The ROC5 plot for the performance of the methods is shown in Figure 6.3. The performance of all methods drops with the shorter sequence length. This is not surprising because of the lower information content of the shorter sequences. Especially for sequence pairs with low sequence identity we would expect a performance drop compared to the full length sequences. However, some methods manage short sequences better than others. Smith-Waterman alignments perform best, BLAST performance is slightly inferior. MMseqs performance improves in comparison to the remaining methods with the decreasing sequence length. In the benchmark with full length query sequences, MMseqs performance with the sensitivity setting 4 is as good as UBLAST, and at half height between BLAST and USEARCH with the sensitivity setting 7. With decreasing length of the query sequences, MMseqs' performance improves compared to UBLAST and RAPsearch and the distance to the best methods Smith-Waterman alignment and BLAST decreases. By contrast, the performance of UBLAST drops dramatically, falling even below the performance of RAPsearch for the shortest 50 length query sequences.

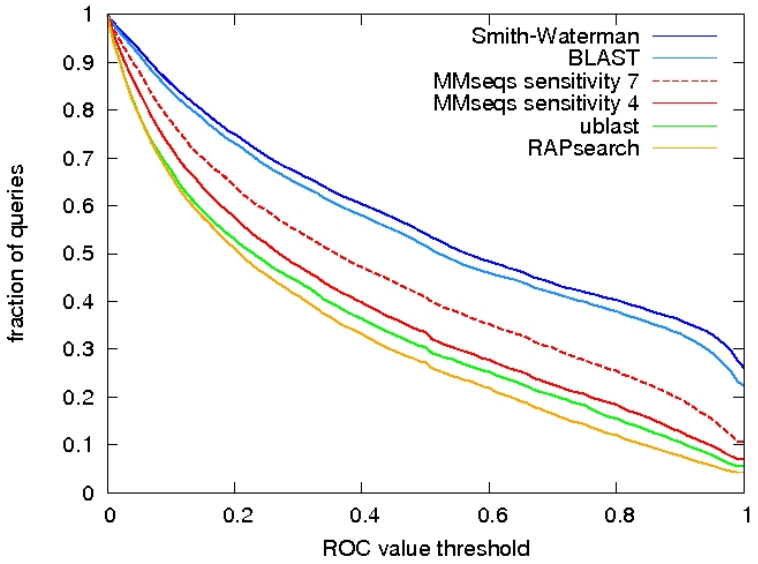
The reason for this difference could be that MMseqs uses similar k-mers while USEARCH and RAPsearch use exact matches for detecting similar sequences. Short sequences offer less possibilities for exact k-mer matches and could explain the fast drop of the performance of USEARCH and RAPsearch on short reads compared to MMseqs.

The performance results in the ROC plot look similar to the ROC5 plot (Figure 6.4). Smith-Waterman alignments and BLAST are the best methods, MMseqs following. Here

is also noticeable that the performance of UBLAST drops considerably with shorter query sequence length.

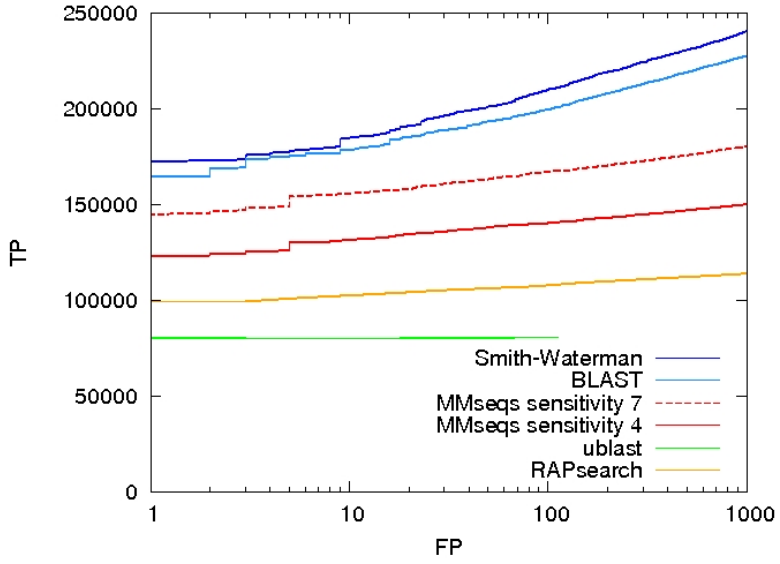


(a)

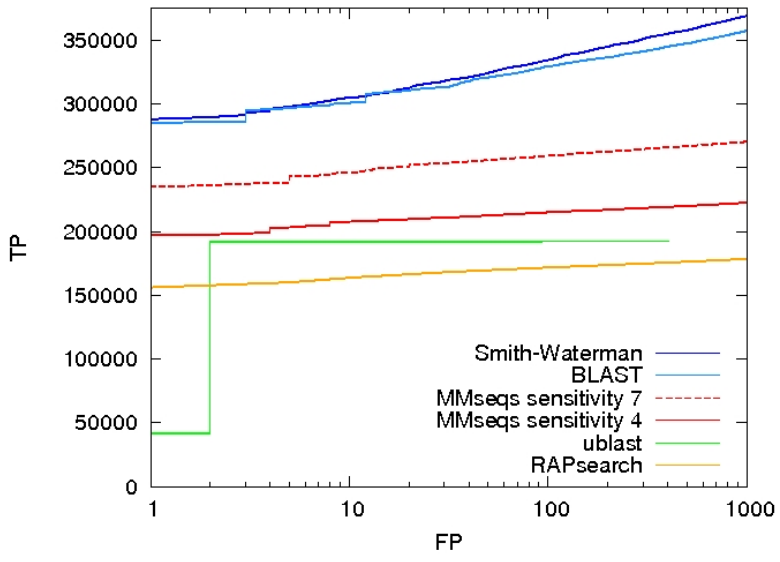


(b)

Figure 6.3.: **ROC5 plots for the performance of different sequence search methods on two short reads datasets. a)** query sequences length 50 amino acids, **b)** query sequences length 100 amino acids. The performance of all methods drops with the shorter sequence length (cf. also ROC5 plot for the benchmark with full length sequences in Figure 6.2). MMseqs shows quite stable performance on short reads. UBLAST is the least robust method, its performance drops the most compared to the performance on full length sequences.



(a)



(b)

Figure 6.4.: **ROC plots for the performance of different sequence search methods on two short reads datasets. a)** query sequences length 50, **b)** query sequences length 100. The results for query sequences are ordered by the score and the number of true positives (y axis) is plotted against the number of false positives (x axis). The reason for the jump in the UBLAST ROC curve is a bug in UBLAST causing sometimes wrong assignment of the sequence headers to the sequences. The UBLAST ROC curve is shorter because of the overall lower number of discovered hits.

6.2.4. Speed benchmark for sequence searches

The search quality should be considered in context of the search time needed by single tools. Since the database used for the quality benchmarks is too small to test for speed, we ran searches with the same query set but using, this time, the whole UniProtKB as the database (UniProtKB version containing 54 790 250 sequences). The time required by each tool is shown in Table 6.1. The time for the index structure generation is given separately from the search time. MMseqs is the fastest tool, being 950 times faster than BLAST with sensitivity 4 (default setting) and 518 times faster than BLAST with sensitivity 7.

6.3. Protein clustering

6.3.1. Benchmarked methods and parameters

We benchmarked the ability of MMseqs, blastclust, CD-HIT, kClust and USEARCH to cluster sequences based on global similarity. Clustering quality runs were made on a computer with 16 cores (processors: Intel Xeon E5-2680, 2.70GHz) and 128 GB RAM.

MMseqs We used the clustering workflow for calculating the clustering of the database. We tested simple and cascaded (option `--cascaded`) clustering each with sensitivity 4 and 7 (`-s 4` and `-s 7`) respectively.

blastclust Blastclust is the clustering software in the BLAST package. We set the length coverage threshold to 0.8 using `-L 0.8` and the score threshold to 0.3 using `-S 0.3`.

CD-HIT We clustered the dataset with CD-HIT down to a sequence identity of 40%, the lowest possible value. We used the `-n 2` setting (k-mer word length) recommended for a clustering threshold of 40%. We set the minimum alignment coverage of the longer sequence to 80% with the `-aL 0.8` option and the number of threads used for the calculation to 16 with the `-T 16` option.

kClust We ran kClust with default parameters, therefore we cluster the database down to 30% sequence identity. Since kClust is single-threaded, we did not use any parallelization options.

6.3.2. Clustering results

We used the same dataset as for the sequence search benchmark, combining all SCOP25 queries and homologous sequences obtained from UniProtKB in one dataset. The resulting database contained 291 022 sequences that could be arranged into 1 118 clusters, based on the SCOP fold assignment. However, since all clustering methods tested here are based on

	time index structure build	time search
MMseqs s=4	1h 17m	6m
MMseqs s=7	1h 17m	11m
swipe	36m	2d 5h 34m
BLAST	36m	3d 23h
UBLAST	1h 07m	48m
RAPsearch	2h 11m	5h 15m

Table 6.1.: Time each tool needed for searching all SCOP25 sequences against the whole UniProtKB.

	#clusters	#seqs per cluster	#corrupted clusters	time
MMseqs s=4 greedy clustering	85 780	3.4	1	4m 03s
MMseqs s=4 set cover	60 915	4.7	1	4m 03s
MMseqs s=4 cascaded	41 173	7.0	3	3m 35s
MMseqs s=7 greedy clustering	41 572	7.0	3	9m 26s
MMseqs s=7 set cover	29 801	9.7	2	9m 26s
MMseqs s=7 cascaded	22 541	12.9	1	5m 07s
blastclust	21 890	13.3	1	7h 58m
CD-HIT	114 386	2.5	260	1h 25m
kClust	91 681	3.2	1	9m 57s
USEARCH	157 981	1.8	11	45s

Table 6.2.: Clustering results on the protein database consisting of SCOP25 and related UniProtKB sequences. Sequences put into the same cluster, but stemming from different folds are considered to be false positives.

pairwise sequence comparisons, they were able to recover only a fraction of the similarities. We considered a cluster as corrupted if it contains sequences from different folds. The results from this clustering procedure for different clustering methods are shown in Table 6.2.

As deduced from Table 6.2, almost all methods produced clusters of high quality with a negligible number of sequences assigned to them by mistake, apart from CD-HIT, which produced 260 corrupted clusters. Blastclust produced the lowest number of 21 890 clusters, requiring about 8 hours. Out of the remaining methods, MMseqs cascaded clustering with sensitivity 7 yielded a low number of 22 541 clusters, requiring about 5 minutes. MMseqs cascaded clustering with sensitivity 4 (default setting for clustering very large databases) gave 41 173 clusters, taking about 3,5 minutes. Cascaded clustering performs in general better than the simple clustering. Both cascaded clustering and default simple clustering in MMseqs uses "set cover" as the clustering algorithm.

To better evaluate the the performance of "set cover" and the greedy clustering algorithms we ran MMseqs using both options, since most popular methods (CD-HIT, USEARCH and kClust) utilize the greedy algorithm. This algorithm takes the longest sequence and puts it and all similar sequences in the first cluster. Then, the next longest sequence and its matches

are put in the second cluster and so on, until no sequences are left. Table 6.2 demonstrates that set cover drastically improves the clustering results for the same alignment results. The reason why so many methods use the greedy clustering algorithm is that an all-against-all comparison of the sequences in the database is required to run set cover. By contrast, the greedy clustering only compares each query sequence to a much smaller representative sequence database.

The remaing three methods CD-HIT, kClust and USEARCH produce significantly larger amount of clusters. CD-HIT is quite slow, requiring 1h 25 min, and it can define only 114 386 clusters. Usearch is very fast, requiring less than one minute for the clustering, but producing mostly singletons, giving a total of 157 981 clusters. kClust, the predecessor method of MMseqs, takes about 10 minutes and produces considerably more clusters than MMseqs, discovering 91 681 clusters.

In general, MMseqs has clearly the best tradeoff between speed and clustering sensitivity. When used with sensitivity 7, it produces only 2% more clusters than BLAST, retaining the same accuracy as BLAST, while being 96x faster than BLAST. Usearch is the only method that is faster than MMseqs, but it produces 7 times more clusters than MMseqs.

6.3.3. UniProtKB clustering

Only two clustering tools are able to cluster very large databases as UniProtKB to sequence identities of lower than 50%: MMseqs and USEARCH. We ran the clustering of UniProtKB (UniProtKB version containing 54 790 250 sequences) on a computer with 32 cores (processors: Intel Xeon CPU E5-4620, 2.20GHz) and 512 GB RAM. MMseqs is able to use all 32 cores for the clustering procedure, while USEARCH is able to only use one core.

We use MMseqs cascaded clustering workflow with default settings to evaluate the clustering procedure.

In USEARCH, we set the lowest sequence identity of clusters to 50%, since it is the lowest recommended value corresponding to the documentation (option `--id 0.5`). We only want to have sequences with pairwise global similarity in one cluster, so we set the query and target sequence coverage in USEARCH to 0.8 using the options `-query_cov 0.8` and `-target_cov 0.8`.

BLAST is much too slow to cluster the UniProtKB database. We estimated the runtime of BLAST clustering using a BLAST run with a small query set against the whole UniProtKB database, as performed in the section 6.2.4, and extrapolated the measured runtime to the clustering of the whole UniProtKB database using an all-against-all comparison. We calculated that clustering based on all-against-all BLAST using all 32 cores would need about 58 years.

MMseqs requires 8 days and 17 hours and 118 G of memory for the clustering procedure. It produces 6 374 156 clusters, i. e. an average of 8,5 sequences per cluster. Usearch, on the other hand, requires, for the same job, 11 days and 2 hours and 42 GB of memory, while

	time	#clusters
BLAST	58y	?
MMseqs	8d 17h	6 374 156
USEARCH	11d 2h	9 822 910

Table 6.3.: Time and number of clusters for BLAST, MMseqs and USEARCH. BLAST time is estimated.

it produces 9 822 910 clusters, i. e. an average of 5,5 sequences per cluster. The results are shown in Table 6.3.

Noteworthy is the small difference in the runtimes between USEARCH and MMseqs, although MMseqs uses all 32 cores of the computer and USEARCH is single-threaded. There are two reasons behind this observation: First, MMseqs calculates all-against-all sequence comparisons, while USEARCH uses the CD-HIT clustering algorithm and compares the query only to the representative sequences determined in each step. In addition, USEARCH accepts the first match of a query to a representative sequence and then aborts the calculation for the query. The process of clustering shows that the clustering becomes ever slower: the first third of the database is clustered on the first day of computation and the remaining 10 days are needed for the remaining two thirds of the database.

The second reason why the runtime of USEARCH is comparable to MMseqs is that USEARCH sequence comparison algorithm is less complex, counting only exact 5-mer matches while MMseqs has to generate and match a list of similar k-mers for each query sequence position.

7. Conclusion and outlook

In recent years, the amount of protein data has increased rapidly and there is a great need for fast and sensitive tools for sequence searching and clustering. Currently, no tool is both able to cluster a large sequence database containing tens of millions of sequences within days and is sensitive enough to discover sequence similarities down to 30%.

We developed a very fast and sensitive protein search and clustering tool called MMseqs. MMseqs is able to cluster the UniProtKB version containing about 54 millions of sequences within 8 days down to 20-30% sequence identity. MMseqs can also search for similar protein sequences in large protein sequence databases containing tens of millions of sequences, discovering sequence similarities of about 20-30%.

With the sensitivity parameter, the user can adjust the trade off between speed and sensitivity of the search and the clustering. At the default sensitivity 4, MMseqs finds more homologs than the other fast protein search methods UBLAST and RAPsearch, while being an order of magnitude faster. At sensitivity 7, setting the sensitivity improves the clustering considerably at the cost of two-fold runtime increase. Generally, MMseqs is about 500 to 1000 times faster than BLAST depending on the settings.

In our benchmark with homologous sequences with 30-80% pairwise sequence identity, MMseqs achieves clustering sensitivities similar to BLAST, while running about 100x faster than BLAST. No other benchmarked method is able to produce comparable results with the same runtime - sensitivity tradeoff. MMseqs is the only tool apart from USEARCH which is able to cluster the UniProtKB database, being slightly faster than USEARCH and producing $\frac{2}{3}$ of the number of clusters.

MMseqs contains three modules: prefiltering, alignment and clustering. It is parallelized in two ways using OpenMP multi-core parallelization and using vectorization with SIMD instructions. It also scales very well on multiple CPUs. Therefore, MMseqs is able to make use of the trend for increasing number of cores per computer that will certainly continue during the next years.

MMseqs offers the possibility to update an already clustered database with new sequences very fast, without the need to run the whole clustering from scratch. This is an important feature with respect to the growth of the UniProtKB which can easily accumulate several hundreds of thousands new sequences per week. A large database containing millions of sequences can be updated with hundreds of thousands of new sequences within hours, making frequent updating possible. Cluster identifiers are kept stable during the updating.

The cascaded clustering implemented in MMseqs improves the clustering results further.

Especially for large databases, cascaded clustering offers a possibility to increase the sensitivity and the maximum cluster size considerably while keeping the memory consumption at the same level and even reducing the runtime.

The next milestone we plan is the development of a more sensitive core algorithm, to further improve the sensitivity of protein searches without speed loss. Furthermore, we will develop profile-based searches which boost the sensitivity considerably at the same speed. We can also easily extend MMseqs to run in parallel on multiple computers in a computer cluster, therefore accelerating it even further. We also plan to apply MMseqs to metagenomic data, e. g. gut microbiomes for medical research, soil for agriculture etc. by searching the 6-frame translated hypothetical protein sequences against the UniProtKB. MMseqs also implements a nucleotide sequence search that yet remains to be tested on real data.

Part II.

Appendix

A. Introduction to ffindex

For the management of the prefiltering, alignment and clustering results, we use ffindex, a simple database for handling huge amounts of small files, developed by Andreas Hauser.

During the development of the MMseqs predecessor clustering method kClust, we noticed that the steadily growing amount of the alignment files containing multiple sequence alignments of the sequences in a cluster has to be written to disc, what overchallenged the file system. Even by dividing the alignment files into different directories we obtained too many alignment files per directory after the further database growth. Also the access to a particular alignment was very complicated, since we had to use a certain file naming convention in order to distribute alignment files evenly over the directories. Moreover, we used too much disc space for the files, since most alignments are very small, but there is always a minimum of disc space reserved for a file. Similar problems occurred with HMM files in HHblits. Moreover, since we planned to implement modular design of MMseqs, we also needed a storage for the intermediate results stemming from a module that allows fast and efficient access to the single results.

An ffindex database consists of an index file and a data file and is implemented in C programming language. Figure A.1 shows the structure of the ffindex database. The data file contains all the datasets stored successively, separated by \0 character, what facilitates the reading of the dataset in C. The index file contains a short alphanumeric key for each

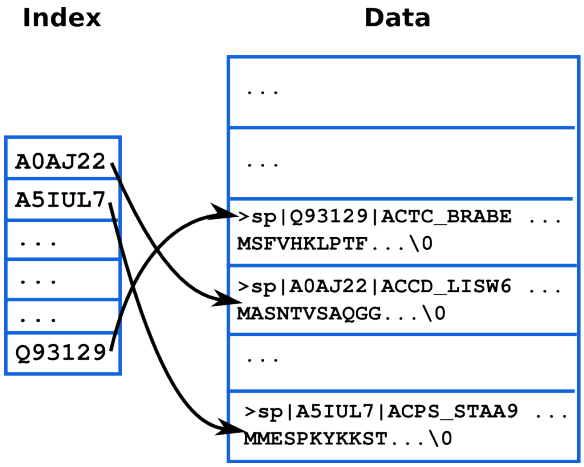


Figure A.1.: **ffindex database structure.** An ffindex database constists of an index file and a data file. The index file is sorted by the key alphabetically and contains the offset and the length of the corresponding dataset.

dataset and the offset and length of this dataset. The index file is sorted. ffindex loads the index file into memory and uses Linux `mmap` function to map the data file into memory. `mmap` loads only the chunks of the file that fit into memory and adjusts the mapping if different chunks are requested or the mapped size gets too big. When a dataset with a certain key is requested, ffindex does a binary search in the sorted index and retrieves the dataset with the help of the offset and length values recorded in the index.

Apart from providing a library for the dataset storage and retrieving, ffindex also offers different binaries for data access and manipulation. `ffindex_build` builds an ffindex database from multiple files or appends new entries. `ffindex_get` can be used for accessing a dataset with a certain key. `ffindex_apply_mpi` executes some program able to read `stdin` and write to `stdout` on each dataset in an ffindex database in parallel on multiple CPUs of a computer and writes the result to a new ffindex database.

We also wrote two binaries to convert FASTA to ffindex format and back. `fasta2ffindex` generates ffindex index and data files for the sequence and header data from a FASTA formatted sequence database. We also provide `ffindex2fasta` binary that converts an ffindex database to FASTA-style formatted flat file.

We use the ffindex library in MMseqs for storing and accessing datasets. MMseqs stores the sequence and header data, prefiltering, alignment and clustering results in ffindex databases. ffindex proved to be very fast for our purposes. We recommend to use ffindex for fast access to the single data records. In addition, ffindex library can be integrated in programs for fast access and processing of the datasets. Finally, datasets contained in huge ffindex databases can be converted and processed very fast by using `ffindex_apply_mpi`. The results are then stored in another ffindex database.

Bibliography

1000 Genomes Project Consortium, G. R. Abecasis, D. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin, R. A. Gibbs, M. E. Hurles, and G. A. McVean. A map of human genome variation from population-scale sequencing. *Nature* **2010**;*467*(7319):1061–1073.

A. Alexeyenko, I. Tamas, G. Liu, and E. L. L. Sonnhammer. Automatic clustering of orthologs and inparalogs shared by multiple proteomes. *Bioinformatics* **2006**;*22*(14):e9–e15.

S. F. Altschul and W. Gish. Local alignment statistics. *Method Enzymol* **1996**;*266*:460–480.

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol* **1990**;*215*(3):403–410.

S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res* **1997**;*25*(17):3389–3402.

R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L. S. Yeh. UniProt: the Universal Protein knowledgebase. *Nucleic Acids Res* **2004**;*32*(Database issue):D115–D119.

E. Bao, T. Jiang, I. Kaloshian, and T. Girke. SEED: efficient clustering of next-generation sequences. *Bioinformatics* **2011**;*27*(18):2502–2509.

B. Berger, J. Peng, and M. Singh. Computational solutions for omics data. *Nat Rev Genet* **2013**; *14*(5):333–346.

S. E. Brenner, P. Koehl, and M. Levitt. The astral compendium for protein structure and sequence analysis. *Nucleic Acids Res* **2000**;*28*(1):254–256.

Y. Bromberg and B. Rost. SNAP: predict effect of non-synonymous polymorphisms on function. *Nucleic Acids Res* **2007**;*35*(11):3823–3835.

K. Chao, W. R. Pearson, and W. Miller. Aligning two sequences within a specified diagonal band. *Comput Appl Biosci* **1992**;*8*(5):481–487.

T. W. Chen, T. Wu, W. Ng, and W. C. Lin. DODO: an efficient orthologous genes assignment tool based on domain architectures. Domain based ortholog detection. *BMC Bioinformatics* **2010**; *11*(Suppl 7):S6+.

D. Chubb, B. R. Jefferys, M. J. Sternberg, and L. A. Kelley. Sequencing delivers diminishing returns for homology detection: implications for mapping the protein universe. *Bioinformatics* **2010**;*26*(21):2664–2671.

V. Chvatal. A greedy heuristic for the set-covering problem. *Math Oper Res* **1979**;*4*(3):233–235.

M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure* **1978**;*5*(suppl 3):345–351.

T. Z. DeSantis, K. Keller, U. Karaoz, A. V. Alekseyenko, N. N. Singh, E. L. Brodie, Z. Pei, G. L. Andersen, and N. Larsen. Simrank: Rapid and sensitive general-purpose k-mer search tool. *BMC Ecol* **2011**;*11*(1):11+.

S. R. Eddy. Hidden Markov models. *Curr Opin Struc Biol* **1996**;*6*(3):361–365.

S. R. Eddy. Accelerated Profile HMM Searches. *PLoS Comput Biol* **2011**;*7*(10):e1002195+.

R. C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics* **2010**; *26*(19):2460–2461.

A. J. Enright, S. V. Dongen, and C. A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Res* **2002**;*30*(7):1575–1584.

M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* **2007**;*23*(2):156–161.

L. Fu, B. Niu, Z. Zhu, S. Wu, and W. Li. CD-HIT: accelerated for clustering the next-generation sequencing data. *Bioinformatics* **2012**;*28*(23):3150–3152.

M. Hauser, C. Mayer, and J. Soding. kClust: fast and sensitive clustering of large protein sequence databases. *BMC Bioinformatics* **2013**;*14*(1):248+.

E. C. Hayden. Is the $1,000 genome for real? **2014**.

A. Heger, S. Mallick, C. Wilton, and L. Holm. The global trace graph, a novel paradigm for searching protein sequence databases. *Bioinformatics* **2007**;*23*(18):2361–2367.

S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *P Natl Acad Sci USA* **1992**;*89*(22):10915–10919.

D. H. Huson and C. Xie. A poor man's BLASTX–high-throughput metagenomic protein database search using PAUDA. *Bioinformatics* **2014**;*30*(1):38–39.

S. D. Kahn. On the future of genomic data. *Science* **2011**;*331*(6018):728–729.

A. Kaznadzey, N. Alexandrova, V. Novichkov, and D. Kaznadzey. PSimScan: algorithm and utility for fast protein similarity search. *PloS One* **2013**;*8*(3).

J. J. Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Res* **2002**;*12*(4):656–664.

M. Kircher and J. Kelso. High-throughput DNA sequencing–concepts and limitations. *BioEssays* **2010**;*32*(6):524–536.

D. C. Koboldt, K. M. Steinberg, D. E. Larson, R. K. Wilson, and E. R. Mardis. The Next-Generation Sequencing Revolution and Its Impact on Genomics. *Cell* **2013**;*155*(1):27–38.

A. Krause, J. Stoye, and M. Vingron. The SYSTERS protein sequence cluster set. *Nucleic Acids Res* **2000**;*28*(1):270–272.

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

H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* **2009**;*25*(14):1754–1760.

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

L. Li, C. J. Stoeckert, and D. S. Roos. OrthoMCL: Identification of Ortholog Groups for Eukaryotic Genomes. *Genome Research* **2003**;*13*(9):2178–2189.

W. Li, L. Fu, B. Niu, S. Wu, and J. Wooley. Ultrafast clustering algorithms for metagenomic sequence analysis. *Brief Bioinform* **2012**;*13*(6):656–668.

W. Li and A. Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics* **2006**;*22*(13):1658–1659.

W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics* **2001**;*17*(3):282–283.

W. Li, L. Jaroszewski, and A. Godzik. Sequence clustering strategies improve remote homology recognitions while reducing search times. *Protein Eng* **2002**a;*15*(8):643–649.

W. Li, L. Jaroszewski, and A. Godzik. Tolerating some redundancy significantly speeds up clustering of large protein databases. *Bioinformatics* **2002**b;*18*(1):77–82.

L. Lo Conte, B. Ailey, T. J. Hubbard, S. E. Brenner, A. G. Murzin, and C. Chothia. Scop: a structural classification of proteins database. *Nucleic Acids Res* **2000**;*28*(1):257–259.

Y. Loewenstein, D. Raimondo, O. Redfern, J. Watson, D. Frishman, M. Linial, C. Orengo, J. Thornton, and A. Tramontano. Protein function annotation by homology-based inference. *Genome Biol* **2009**;*10*(2):207+.

C. Mayer. Fast method fro sequence comparison and application to database clustering. *Master thesis* **2007**;.

M. L. Metzker. Sequencing technologies - the next generation. *Nat Rev Genet* **2010**;*11*(1):31–46.

V. Miele, S. Penel, and L. Duret. Ultra-fast sequence clustering from similarity networks with SiLiX. *BMC Bioinformatics* **2011**;*12*(1):116+.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol* **1970**;*48*(2):443–453.

T. Nepusz, R. Sasidharan, and A. Paccanaro. SCPS: a fast implementation of a spectral method for detecting protein families on a genome-wide scale. *BMC Bioinformatics* **2010**;*11*(1):120+.

V. H. Nguyen and D. Lavenier. PLAST: parallel local alignment search tool for database comparison. *BMC Bioinformatics* **2009**;*10*(1):329+.

J. Park, L. Holm, A. Heger, and C. Chothia. RSDB: representative protein sequence databases have high information content. *Bioinformatics* **2000**;*16*(5):458–464.

W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *P Natl Acad Sci USA* **1988**;*85*(8):2444–2448.

S. Powell, D. Szklarczyk, K. Trachana, A. Roth, M. Kuhn, J. Muller, R. Arnold, T. Rattei, I. Letunic, T. Doerks, L. J. Jensen, C. von Mering, and P. Bork. eggNOG v3.0: orthologous groups covering 1133 organisms at 41 different taxonomic ranges. *Nucleic Acids Res* **2012**;*40*(Database issue):D284–D289.

J. Qin, R. Li, J. Raes, M. Arumugam, K. S. S. Burgdorf, C. Manichanh, T. Nielsen, N. Pons, F. Levenez, T. Yamada, D. R. Mende, J. Li, J. Xu, S. Li, D. Li, J. Cao, B. Wang, H. Liang, H. Zheng, Y. Xie, J. Tap, P. Lepage, M. Bertalan, J. M. Batto, T. Hansen, D. Paslier, A. Linneberg, B. B. Nielsen, E. Pelletier, P. Renault, T. Sicheritz-Ponten, K. Turner, H. Zhu, C. Yu, S. Li, M. Jian, Y. Zhou, Y. Li, X. Zhang, S. Li, N. Qin, H. Yang, J. Wang, S. Brunak, J. Doré, F. Guarner, K. Kristiansen, O. Pedersen, J. Parkhill, J. Weissenbach, MetaHIT Consortium, P. Bork, D. D. Ehrlich, and J. Wang. A human gut microbial gene catalogue established by metagenomic sequencing. *Nature* **2010**;*464*(7285):59–65.

N. Rappoport, S. Karsenty, A. Stern, N. Linial, and M. Linial. ProtoNet 6.0: organizing 10 million protein sequences in a compact hierarchical family tree. *Nucleic Acids Res* **2012**;*40*(Database issue):D313–D320.

T. Rattei, P. Tischler, S. Götz, M.-A. Jehl, J. Hoser, R. Arnold, A. Conesa, and H.-W. Mewes. SIMAP—a comprehensive database of pre-calculated protein sequence similarities, domains, annotations and clusters. *Nucleic Acids Res* **2010**;*38*(suppl 1):D223–D226.

M. Remm, C. E. Storm, and E. L. Sonnhammer. Automatic clustering of orthologs and in-paralogs from pairwise species comparisons. *Journal of molecular biology* **2001**;*314*(5):1041–1052.

M. Remmert, A. Biegert, A. Hauser, and J. Söding. HHblits: lightning-fast iterative protein sequence searching by HMM-HMM alignment. *Nat Methods* **2012**;*9*(2):173–175.

T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics* **2011**;*12*(1):221+.

T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* **2000**;*16*(8):699–706.

B. Rost and C. Sander. Prediction of protein secondary structure at better than 70% accuracy. *J Mol Biol* **1993**;*232*(2):584–599.

D. B. Rusch, A. L. Halpern, G. Sutton, K. B. Heidelberg, S. Williamson, S. Yooseph, D. Wu, J. A. Eisen, J. M. Hoffman, K. Remington, K. Beeson, B. Tran, H. Smith, H. Baden-Tillson, C. Stewart, J. Thorpe, J. Freeman, C. Andrews-Pfannkoch, J. E. Venter, K. Li, S. Kravitz, J. F. Heidelberg, T. Utterback, Y. H. Rogers, L. I. Falcón, V. Souza, G. Bonilla-Rosso, L. E. Eguiarte, D. M. Karl, S. Sathyendranath, T. Platt, E. Bermingham, V. Gallardo, G. Tamayo-Castillo, M. R. Ferrari, R. L. Strausberg, K. Nealson, R. Friedman, M. Frazier, and C. C. Venter. The Sorcerer II Global Ocean Sampling expedition: northwest Atlantic through eastern tropical Pacific. *PLoS Biol* **2007**;*5*(3):e77+.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol* **1981**;*147*(1):195–197.

J. Söding. Protein homology detection by hmm-hmm comparison. *Bioinformatics* **2005**;*21*(7):951–960.

J. Söding, A. Biegert, and A. N. Lupas. The HHpred interactive server for protein homology detection and structure prediction. *Nucleic Acids Res* **2005**;*33*(Web Server issue):W244–W248.

J. Söding and A. N. Lupas. More than the sum of their parts: On the evolution of proteins from peptides. *Bioessays* **2003**;*25*(9):837–846.

E. L. Sonnhammer, S. R. Eddy, E. Birney, A. Bateman, and R. Durbin. Pfam: Multiple sequence alignments and HMM-profiles of protein domains. *Nucleic Acids Res* **1998**;*26*(1):320–322.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr Dobb's Journal* **2005**;pages 1–9.

B. Suzek, H. Huang, P. McGarvey, R. Mazumder, and C. H. Wu. UniRef: comprehensive and non-redundant UniProt reference clusters. *Bioinformatics* **2007**;*23*(10):1282–1288.

J. Tan, D. Kuchibhatla, F. L. Sirota, W. A. Sherman, T. Gattermayer, C. Y. Kwoh, F. Eisenhaber, G. Schneider, and S. M. Stroh. Tachyon search speeds up retrieval of similar sequences by several orders of magnitude. *Bioinformatics* **2012**;*28*(12):1645–1646.

R. L. Tatusov, N. D. Fedorova, J. D. Jackson, A. R. Jacobs, B. Kiryutin, E. V. Koonin, D. M. Krylov, R. Mazumder, S. L. Mekhedov, A. N. Nikolskaya, B. S. Rao, S. Smirnov, A. V. Sverdlov, S. Vasudevan, Y. I. Wolf, J. J. Yin, and D. A. Natale. The COG database: an updated version includes eukaryotes. *BMC bioinformatics* **2003**;*4*(1):41+.

C. Trapnell and S. L. Salzberg. How to map billions of short reads onto genomes. *Nat Biotechnol* **2009**;*27*(5):455–457.

A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Comp Appl Biosci* **1997**;*13*(2):145–150.

Y. Ye, J. H. Choi, and H. Tang. RAPSearch: a fast protein similarity search tool for short reads. *BMC Bioinformatics* **2011**;*12*(1):159+.

G. Yona, N. Linial, and M. Linial. ProtoMap: automatic classification of protein sequences and hierarchy of protein families. *Nucleic Acids Res* **2000**;*28*(1):49–55.

S. Yooseph, G. Sutton, D. B. Rusch, A. L. Halpern, S. J. Williamson, K. Remington, J. A. Eisen, K. B. Heidelberg, G. Manning, W. Li, L. Jaroszewski, P. Cieplak, C. S. Miller, H. Li, S. T. Mashiyama, M. P. Joachimiak, C. van Belle, J.-M. Chandonia, D. A. Soergel, Y. Zhai, K. Natarajan, S. Lee, B. J. Raphael, V. Bafna, R. Friedman, S. E. Brenner, A. Godzik, D. Eisenberg, J. E. Dixon, S. S. Taylor, R. L. Strausberg, M. Frazier, and J. C. Venter. The Sorcerer II Global Ocean Sampling Expedition: Expanding the Universe of Protein Families. *PLoS Biol* **2007**;*5*(3):e16+.

Y. Zhao, H. Tang, and Y. Ye. RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. *Bioinformatics* **2012**;*28*(1):125–126.