A highly scalable system for genome based computational microbial diagnostics

Tilo Eißler

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur

A highly scalable system for genome based computational microbial diagnostics

Tilo Eißler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Martin Bichler
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. Arndt Bode
	2. Univ.-Prof. Dr. Burkhard Rost

Die Dissertation wurde am 21. Mai 2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12. September 2012 angenommen.

Abstract

In-silico primer and probe design based on nucleic acid genome sequence data is of great interest in many areas of research and economy concerning microbial diagnostics, i.e. molecular identification and detection of bacteria and viruses. Currently available software tools have reached a performance limit due to the extremely fast growth of publicly available sequence data of microorganisms produced by high throughput sequencing technology.

The aim of this thesis has been to develop a highly scalable integrated system for microbial in-silico diagnostics based on microbial genome sequence data. The first central component is a framework for efficient retrieval and management of sequence and annotation data from different heterogeneous data sources with the ability to obtain data sets ranging from small subsets to all available microbial genomes. To support fast non-heuristic oligonucleotide string matching and applications for primer/probe design on this data, required by computational microbial diagnostics, a new memory independent nucleic acid sequence index structure is developed as the second central component. In order to handle the huge amount of available and impending data, central components of the integrated system have been optimized and parallelized to efficiently utilize multi-core-architectures and high performance computers as both will continue to multiply the number of computing cores to increase computing power in the future. The integrated system will allow a large amount of users to continue their work on desktop computers while also enabling new HPC-based applications and research not possible so far with the currently available software.

Acknowledgment

Many people have contributed to the success of my work in different ways. First of all I would like to thank Prof. Dr. Arndt Bode for providing me the opportunity to do this work at the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) in a supportive working environment. Furthermore I want to thank Dr. Harald Meier for his supervision of my work and the various helpful discussions. From my colleagues at the LRR, I want to give a special thanks to Kai Christian Bader for the time we have been working together, discussing and helping each other when getting stuck at some weird programming issues. I also want to thank the good friend of mine Thomas Döhring for carefully proof-reading my thesis and providing helpful comments. I am also very grateful to the numerous people not mentioned explicitly, contributing to my work in different ways. Such a work would not be possible without helpful colleagues giving some advice from time to time. Finally I want to thank the *Bayerische Forschungsstiftung* for supporting my work within the project Nanobak (grant [AZ 767-07]) as well as the Leibniz-Rechenzentrum of the Bavarian Academy of Sciences and Humanities for giving me the opportunity to finish my thesis.

Contents

1 Introduction

1.1 Motivation

According to the Genome OnLine Database (GOLD) [81] more than 2800 complete bacterial genomes are published and for almost 7000 sequencing is ongoing [45]. The overall genome sequencing efforts are reflected by a high increase of complete and ongoing projects listed by the GOLD project (figure 1.1). These numbers are likely to further grow rapidly in the next few years as the latest sequencing technologies, producing large amounts of molecular sequence data in short time, will soon become standard [85, 90].

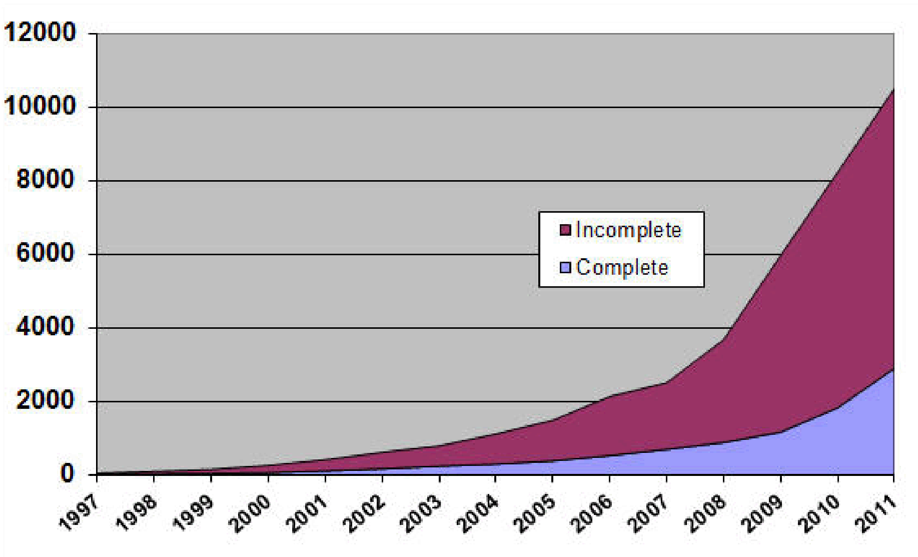


Figure 1.1: Development of genome sequencing project numbers (GOLD October 2011 [45])

This genome sequence data is of great interest concerning analysis such as computational molecular diagnostics (CMD). CMD relies on the in-silico, i.e. performed on a computer, search for molecular markers, primers and probes based on sequence data. It can lead to a faster development of molecular detection methods for pathogens while reducing the experimental cost in the wet lab.

On the computational side two major areas are of great interest in order to conduct CMD. The first area concerns the efficient access and management of the genome sequence and related data from different sources. Second, as directly processing sequence string searches on the data is very compute intensive, especially primer/probe design relies on indexing structures for speeding up string matching. These index structures need to be capable of dealing with the amount of data available.

The development of computer hardware imposes additional barriers for CMD when trying to deal with the amount of genome sequence data available and impending. Although the main memory size doubles every 18 to 24 month, the indexing structures widely employed for CMD already exceed the main memory size of common desktop computers for the genome sequence data available. In addition, as modern processors increase their performance by increasing the number of computational cores, algorithms must leverage parallelism in order to speed up computations of performance critical parts.

Molecular Data Access Prior to conducting extensive analyses it is necessary to build reliable local databases from own experimental as well as reference sequence data collected from the public primary or secondary sequence data sources. The latter is provided by several projects gathering and offering free public access to molecular sequence data like the EMBL-EBI projects EMBL [70], EnsEMBL [34] and GenomeReviews [67] as well as projects from other institutions like DDBJ [131] and GenBank [10] among many others. The different projects employ various schemas and database management systems (DBMS) and thus differ in the way the data is represented and stored. Access is provided in several ways ranging from web-interfaces with export to flat files [10, 34, 41], direct downloads of different flat file packages to database dumps [10, 34, 41] or even direct access to the database servers [34].

The massive increase of the amount of sequence data together with the heterogeneity of the sources require efficient solutions for the data management. According to Jagadish and Olken many home-grown solutions which served well so far do not scale and thus need to be replaced [63]. It is furthermore necessary to map between different data representations as well to allow data interchange. Hence Jagadish and Olken demand a standardized interface for the various available heterogeneous data sources [63].

Existing integrated systems for storing molecular sequence data and identifying molecular markers, primers and probes such as the ARB software environment [84] have major drawbacks. In case of ARB these drawbacks result from the software being more than 15 years old [130]. For example it utilizes a deeply integrated proprietary database management system and the size of databases is limited by the available main memory.

Other sequence data integration approaches are available in form of data warehouses like *BioMart* [122] or database integration systems like *ACNUC* [47] or the *Sequence Retrieval System (SRS)* [36]. These systems are targeted to provide data mining capabilities or to offer read-only data access over web-frontends. Although some provide an application programming interface (API) as well, they do commonly not offer access to local databases.

The different available bioinformatics frameworks like the *Bio** projects (BioJava, BioPerl, BioPython and BioRuby) [87] or the C++ APIs like *Bio++* [31], *Seqan* [30, 44] or the *NCBI C++ Toolkit* [103, 136] do offer access to sequence data in form of flat file importers or to selected genome database schemas. Unfortunately, none offers unified access to the different heterogeneous genome database schemas utilizing different types of DBMS.

For users or application developers this has severe implications. The first option for them is to rely on the data integration platforms. If sequence analysis data from own experiments is involved, this means giving up control by uploading the data and relying on the security and honesty of the service provider. In addition the user is dependent, i.e. if the service is temporarily not available or discontinued, analysis is not possible any more. The second option is to rely on one of the available bioinformatics frameworks. This may include additional inconvenient manual steps to conduct by the user, for example to retrieve and convert required data from different heterogeneous sources prior to carrying out analysis such as primer/probe design on the data.

In-silico Primer/Probe Design Primer and probes are utilized for molecular identification and detection for example of bacteria and viruses. The in-silico design and evaluation of such primers and probes often relies on fast methods for non-heuristic exact and approximate oligonucleotide string matching. In the field of molecular diagnostics based on nucleic acid data, indexing structures are widely utilized to speed up computations [84, 73, 111].

A wide range of different index based solutions exist to design primer and probes based on genome sequences. The *CMD/PSID* is capable of identifying signatures based on sequence collections [74]. Unfortunately it can deal only with single sequences as a target, not whole groups of sequences. This is a major drawback limiting its field of application. Designing primers and probes for detecting and distinguishing organism groups rather than single organisms is required in applications such as microbial population analysis and molecular screening for microbial pathogens or indicators.

Two further approaches provide pre-processed signature collections are the web-based *Insignia* server [110] and *CaSSiS* [6]. Both allow to select candidates based on the clustering information which served as pre-processing input. Designing new primer/probe signature candidates on-the-fly is not possible without repeating the compute intensive pre-processing of the whole collection with different parameters set.

In contrast an approach capable of conducting on-the-fly primer/probe design based on a single or a group of sequences in conjunction with evaluation capabilities is the ARB software environment [84]. It relies on a central index structure supporting molecular diagnostics, the ARB PT-Server.

The ARB PT-Server is a representative example for an in-memory suffix tree based index approach. Deeply integrated into ARB and relying on the ARB database as sequence source, the PT-Server provides approximate oligonucleotide string matching capabilities based on the Hamming-distance metric to identify substituted bases. It supports in-silico primer/probe design and evaluation widely applied in microbiology [4, 119, 68] and serves as the basic component of the *probeCheck* server and the comprehensive signature search tool *CaSSiS* [83, 6].

The PT-Server operates completely in-memory during construction and application. Although it reduces its memory requirements by truncating the suffix tree, it faces severe problems in dealing with the increasing amount of sequence data. Furthermore insertions and deletions (indels) cannot be identified during approximate string matching which would require the utilization of the Levenshtein-distance metric.

The problem of increasing amounts of genome sequence data has been tackled by many state-of-the-art indexing techniques. Although not providing full primer/probe design functionality these techniques, mainly originating from indexing structure theory, are of interest as a core structure for new developments.

The k-truncated suffix tree (*kTST*) [120] limits the height of the suffix tree to reduce its memory requirements like the ARB PT-Server, thus suffering from the same memory constraints. The enhanced suffix array (*eSA*) invented to replace suffix trees, has reduced memory requirements, but still needs to fit into main memory entirely during application [1]. The so called self-indexes, i.e. a compressed representation of the source text in combination with an index structure, require main memory in the order of the size of the original input text [102]. Unfortunately, according to Russo and colleagues, approximate string matching based on self-indexes faces severe slowdowns compared to classical indexes like suffix trees or arrays [116].

Besides the aforementioned in-memory solutions, several approaches try to deal with the large amount of data by utilizing secondary storage during index construction and application. The disk resident suffix arrays (*rSA*) offers moderate disk space requirements as well as providing exact string matching, but no approximate string matching capabilities [92]. A well studied approach are suffix trees in external memory (*eST*), recently reviewed by Barsky and colleagues [8]. They can be constructed with limited main memory efficiently, although memory requirements on secondary storage are high. Furthermore, an open challenge for the existing *eST* approaches is the utilization for approximate string matching. For gene sequence data, C. Hodges conducted a promising preliminary evaluation combining suffix trees on secondary storage with a compression scheme supporting efficient oligonucleotide approximate string matching [55]. Unfortunately it is not capable of dealing with genome sequence data.

As the suffix tree is still an essential index structure despite its high memory consumption and construction efforts, several approaches try to speed up construction by utilizing parallelization of computations, either on shared memory multi-core systems [134] or on cluster computer systems utilizing the Message Passing Interface (MPI) [42, 88, 141].

Existing integrated systems for comprehensive sequence analysis and primer/probe design such as the ARB software environment are relying on indexing structures. As a consequence they need to deal with current challenges. Unfortunately, neither the currently employed nor other existing indexing approaches are able to deal with the demands of providing low main memory requirements in conjunction with enhanced and fast approximate oligonucleotide string matching.

1.2 Scientific Contribution

The scientific contribution of this thesis is divided into three major parts.

First, the *Unified Molecular Data Access (UMDA)* framework has been developed. It comprises a generic object model as well as abstract interfaces for database access and index based primer/probe design and evaluation capabilities. For the database interface a query system allows to select subsets of sequence data entries. A unique plugin system based on the abstract interfaces allows to develop algorithms independently of specific database management systems or index structures. With this the *UMDA* framework significantly eases the development of future proof applications facing the various existing heterogeneous data sources. Furthermore *UMDA* based applications can benefit from future developments simply by adding new plugins.

Second, with *PTPan* a stream-compressed index structure for nucleic acid genome and gene sequence data based on a truncated suffix tree on secondary storage has been developed. It offers enhanced approximate oligonucleotide string matching capabilities based on either the Hamming- or Levenshtein-distance metric. It is capable of operating with limited main memory utilizing secondary storage efficiently. Applications with comprehensive functionality for primer/probe design and evaluation are incorporated within the index. A preliminary version has been published in [32]. Optimization and parallelization of key components have been conducted in order to speed up construction as well as application on shared memory systems. In addition, a construction algorithm relying on MPI enables HPC based analysis of large genome sequence data collections. With *PTPan* it is now possible to index huge nucleic acid gene and genome sequence collections even on systems with limited main memory. The *PTPan* based applications provide good response times, even compared to competing in-memory solutions. *PTPan* clearly relaxes the problems of providing efficient oligonucleotide string matching capabilities facing the enormous growth of molecular sequence data.

Third, with the *UMDA Primer/Probe (UPP) Designer* an integrated system for primer and probe design and evaluation based on the *UMDA* framework has been developed. It allows the utilization of the sequence data selection offered by the *UMDA database interface* query system in conjunction with the primer/probe design and evaluation capabilities of the *UMDA search index interface*, for example based on the *SII* plugin incorporating *PTPan*. Utilizing the flexibility of the *UMDA* framework in conjunction with the sophisticated capabilities of *PTPan*, the *UPP Designer* supports the fast development of molecular detection methods for pathogens. Scientists can now easily conduct primer/probe design processes on either a subset or all microbial nucleic acid genome sequences available utilizing a single software tool.

The source code of *UMDA*, *UPP Designer* and *PTPan* is available at `http://ptpan.lrr.in.tum.de/`. Furthermore, the ARB software environment incorporates *PTPan* `http://www.arb-home.de/`.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: In chapter 2 the relevant background information is provided. This comprises an overview over nucleic acid sequence and its related data. Furthermore the problem fields of string matching and indexing structures are introduced followed by the software related to and employed by this thesis. Finally an overview over modern hardware architectures and optimization techniques is given. In chapter 3 the results of this thesis are presented beginning with the *Unified Molecular Data Access (UMDA)* framework. Afterwards the *PTPan* index and the primer/probe design and evaluation applications relying on it are presented. Finally the software components developed including *UMDA Primer/Probe (UPP) Designer* are described. In chapter 4 the performance evaluation conducted is presented followed by the discussion of the thesis results in chapter 5. Finally chapter 6 and chapter 7 provide a thesis summary and an outlook on future work.

2 Basics And Related Work: Genome Data, Index Structures, Software And Computer Architecture

The following chapter introduces the important basics and the work related to this thesis, starting with a brief overview of nucleic acid sequence and its associated data including the main storage formats as well as some of the most important data providers.

Afterwards an introduction to approximate string matching is given followed by a description of the relevant indexing structures for speeding up approximate oligonucleotide pattern matching in large nucleic acid sequence data collections.

Subsequently the relevant software applications including database management systems and biological data integration platforms, bioinformatic frameworks for nucleic acid sequence analysis and tools for primer and probe design are presented.

Finally a brief overview over the memory hierarchy of modern computer systems and an introduction to parallel computer architectures including program optimization and parallelization techniques and frameworks is given.

2.1 Nucleic Acid Sequence Data

Deoxyribonucleic acid (DNA) and *ribonucleic acid (RNA)* are biological macromolecules essential for life representing the genetic information. In his definitive book "Principles of Nucleic Acid Structure" W. Saenger gives an introduction to their composition and functionality [117].

DNA is the concatenation of so called nucleotides. It carries the genetic information of an organism commonly in two anti-parallel polymer strands arranged as a double helix, except for some viruses. Thereby a single *DNA* strand consists of coding and non-coding regions. Coding regions are copied into the related nucleic acid *RNA*. This process is called transcription. Several types of *RNA* exist. For example the *Messenger RNA (mRNA)* encodes the structure of a protein for his part consisting of amino acids. The mechanism for decoding the *mRNA* into amino acid is provided by the *Ribosomal RNA (rRNA)*. Besides the latter several other types of RNA exist.

2.1.1 Nucleic Acid Sequence Alphabets

A nucleic acid sequence is actually the primary structure of a certain piece of nucleic acid, namely the sequence of its nucleotides. The heterocyclic base is the essential informative component, but it is just one component of a nucleotide. The bases cannot be concatenated on their own. The formal representation of a nucleic acid sequence is a string of characters representing the sequence of nucleotides.

A *DNA* sequence is a concatenation of four different bases only, namely *Adenine*, *Cytosine*, *Guanine* and *Thymine* which are abbreviated as "A", "C", "G" and "T". *RNA* sequences are based on almost the same base characters with only *Thymine* being replaced by *Uracil* (abbreviated as "U"). Several other characters are utilized to represent ambiguous positions, for example "N" for any base or "R" for an "A" or "G".

The nucleotide base sequence code names have been standardized by the International Union of Pure and Applied Chemistry (IUPAC), an international federation which works on standardizing nomenclature in chemistry and other related science [62].

Within this thesis, two different *DNA/RNA* alphabet definitions are distinguished. The *DNA4* alphabet consists of the four symbols "A", "C", "G" and "T" (or "U" in RNA respectively). The *DNA5* alphabet adds the symbol "N" which stands for all non-*DNA4* symbols. This is important to represent the ambiguous bases which are commonly present in many nucleic acid sequences in public repositories.

2.1.2 Nucleic Acid Sequence Related Data

Besides the nucleic acid sequences itself there are different kinds of related data mostly derived from sequence analysis.

Sequence order and direction The direction of a nucleic acid sequence is read from the so called 5′ (five prime) end to the 3′ (three prime) end per convention. The naming is derived from the structure of the nucleic acids, i.e. the direction in which it is synthesized by polymerase. This direction is often referred to as "forward direction", the *reverse* order as "reverse direction" (3′ to 5′).

For a double-stranded DNA, one of the two strands is considered the sense strand and the other as antisense strand, i.e. the *complementary* sequence to the sense strand. The sense strand is normally the one that has the same sequence as the *mRNA*. Therefore the strand that serves as template for *mRNA* is the antisense strand.

Genome annotations In order to enrich a newly sequenced genome with information about its functional regions, the sequence is annotated. This process is carried out manually, semi-automated or automated and produces sequence features. The aim is to "identify the key features of the genome - in particular the genes and their product" [128]. Despite the latter, sequence features comprise also the functional and regulatory elements in the non-coding regions. On the sequence string, each *gene* is the name for a coding region, also named *coding sequence (CDS)*, which is identified by one or more locations comprising a start and end index. Annotations are stored along with the sequence data in the genomic sequence databases (refer to section 2.1.5).

Signatures, primers and probes Many biological applications rely on DNA/RNA *signatures*, i.e. short nucleic acid sequences. A signature is representative for a longer nucleic acid sequence. Therefore the detection of its presence can stand for the detection of the whole piece of nucleic acid, i.e. a gene or even a whole genome. A signature can characterize a function but sometimes even an organism or a group of organisms.

Depending on the biological application, these signatures can be used to design primers for *polymerase chain reaction (PCR)* or probes for organism detection, for example by *fluorescence in-situ hybridization (FISH)*. Primer/probes are the reverse complement of a signature.

For signatures respectively primers/probes, *sensitivity* and *specificity* are distinguished. Sensitivity, often denoted as coverage as well, is a measure for how many of the targeted genes or genomes are covered by a signature. If the specificity of a signature is high, the number of out-group sequences hit is minimal or at least a high distance is guaranteed.

More information on designing and evaluating primers and probes will be given later in section 2.4.4.1.

Alignment Another form of sequence analysis is the alignment of two or more sequences. The aim is to identify regions of similarity. These similar regions can be a hint to show evolutionary relationships between sequences respectively their related organisms [54]. In order to allow the unambiguous representation of similarities as well as differences of equal or unequal length sequences, the sequences are arranged vertically inserting gap characters, briefly called gaps. These gaps are denoted by a hyphen ("-"). Some tools utilize dots (".") as well, for example the ARB software (refer to section 2.4.4.2).

Alignments can be represented in form of a *'Compact Idiosyncratic Gapped Alignment Report' (CIGAR)* format string [124] or its extended version [78]. The CIGAR format stores series of `<operation, length>` pairs. Operations are match, insertion or deletion. The length denotes how often this operation is repeated. The gapped alignment representation can be reconstructed from the sequence string and its corresponding CIGAR string. Figure 2.1 shows two aligned sequences and their respective CIGAR strings.

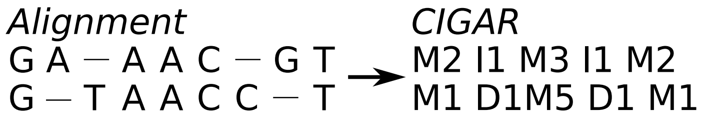


Figure 2.1: Nucleic acid sequence alignment example for two sequences as well as its CIGAR representation

Differential alignment In a *differential alignment* one sequence stretch functions as reference. One or more other stretches are not shown as a string of DNA bases, but as a sequence of characters denoting identical bases or differences. In case of an identical base a equal sign ("=") is printed while for a substitution the base of the aligned stretch is shown. If the aligned stretch contains the wildcard character "N", it is printed like a substitution in order to highlight it. The other two edit operations possible, i.e. insertion and deletion, are symbolized by an asterisk ("*") respectively an underscore ("_"). In case of insertions several neighboring characters may be denoted by a single asterisk to keep the vertical alignment within the size of the reference sequence stretch.

Two examples for differential alignments are shown in figure 2.2. The second example contains an insertion shadowing another symbol in order to keep the vertical alignment of corresponding bases.

```
reference  G A A A C G T
     diff  = T N = = _ =
           G T N A C   T
     diff  = C = = = * =
           G C A A CGGT
```

Figure 2.2: Differential alignment example with a reference and two sequences aligned to it. The grey stretches are the sequences for which the differential alignment was build.

2.1.3 Genome Sequence Database Schemas

There are various different types of biological databases for different kinds of biological data. An overview has been presented by De Francesco and colleagues including a classification for the different types of genomic databases and schemas as well as a list of example instances [28]. The term *database* on its own is thereby referring to a logical collection of data while the term *database schema* refers to the structure of the database.

In the following some widely used nucleic acid genome sequence database schemas are presented, although the list is not exhaustive. The schemas rely on different database management systems which are described later in chapter 2.4.1.

ARBDB The *ARB database (ARBDB)* schema is part of the ARB software environment and tightly coupled with the ARB database management system [84]. It is an integrated hierarchically structured database schema for storing sequence data and its related information in individual database fields. The schema differs between two types of databases, gene and genome. For both reference and taxonomy is included among other meta-data. For genomes, feature information is included in addition.

BioSQL The *BioSQL* database schema is a joint effort of the projects participating in the Open Bioinformatics Foundation (OBF) [106]. It provides a generic relational model for storing sequence data along with features, annotation for both, literature references and taxonomy among other data. The schema is available for different relational database management systems, for example MySQL and PostgreSQL [14].

Chado The *Chado* relational database schema is part of the GMOD project available for PostgreSQL [96]. Chado is a generic approach capable of storing molecular sequence data along with annotation data, for example features, publications and phylogenetic information among other data.

EnsEMBL The *EnsEMBL* genome database project provides a relational database schema as part of their bioinformatics framework [57]. The *EnsEMBL* core schema is capable of storing nucleic acid and assembly sequences, computed features and genes along with other miscellaneous information [125]. The SQL schema is available for MySQL and can be downloaded along with MySQL database dumps from the public FTP-server accessible via the EnsEMBL website [34]. The schema is utilized by the *Ensembl genomes* and *Genome Reviews* projects [66, 67].

2.1.4 Text File Data Formats

Besides the database schemas presented in section 2.1.3, each relying on a database management system, there are various text file formats for nucleic acid sequence and feature data. Often they are denoted as flat file formats as well.

The *FASTA*-file format contains one or more bare sequence data entries besides some meta-data in a header for each entry. This header may differ in its composition depending on the bioinformatics software utilized, although it often contains a sequence identifier. A *FASTA*-file with more than one sequence is called *multiFASTA*-file as well.

With *SeqXML* a simple XML based format for sequence data was recently proposed by Schmitt and colleagues [118]. It allows to store the same information as *FASTA*-files but with the meta-data in a standardized way.

Besides the simple sequence file formats there is the *EMBL-Bank* flat file format defined in the EMBL-Bank User Manual [33]. It comprises sequence data with corresponding feature information as well as meta-data like literature references. Similar flat file formats are also offered by DDBJ [65] and GenBank [9].

Besides the *EMBL-Bank* flat file format there is an *EMBL-Bank XML* file format as well which comprises the same content [33].

2.1.5 Sequence Data Provider

There are numerous nucleic acid sequence data provider which are commonly divided into primary and secondary ones.

The three large primary providers for genome sequence data offer a comprehensive collection of the sequenced genomes available so far. The first is the EMBL EBI with the projects *Ensembl* [57] and the newer *Ensembl genomes* [66]. The other providers are *Genbank* [9] and *DDBJ* [65]. All three collaborate in the *International Nucleotide Sequence Database Collaboration* [60] to provide common sequence data feature representations and standards for annotation practice.

Based on the data provided by the primary sources there are various projects utilizing the genome sequence data to build curated and non-redundant collections of sequences targeting specific purposes, i.e. secondary databases. The *NCBI's Reference Sequence (RefSeq)* database provides whole genome sequences with their related information [114]. Similar, *Genome Reviews* provides comprehensive views of the genomic sequence of organisms [129]. Other projects like *SILVA* [113], *Greengenes* [29] or *RDP* [25] are more specialized. They provide curated gene sequence databases. These are often enriched by some analysis results, for example phylogenetic trees to document the evolutionary relationship.

A comprehensive list of molecular biology databases including many nucleic acid sequence databases has been provided by a recent *Nucleic Acids Research Database Issue* [39].

2.2 String Matching

The next sections will give an introduction to the relevant parts of the problem field of string matching. First string matching is defined. Afterwards the different kinds of string matching being of interest for this work are described.

2.2.1 Problem Definition

Consider $R = r_1r_2...r_N$ being a string with $n = |R|$ symbols over the alphabet Σ. The string M with length $m = |M|$ denotes the search pattern over the same alphabet. It is assumed that $m \ll n$ in the context of this work. With this, *string matching*, also known as *pattern matching*, is the problem of finding one or all occurrences of M in R.

String matching can be further divided into *exact string matching* and *approximate string matching (ASM)*. For exact string matching, all occurrences in R match exactly the pattern M. In contrast, ASM occurrences can have a certain number of differences to M.

2.2.2 Exact String Matching With Wildcards

Like the common application of wildcards within the search pattern M, it is also possible to define a character as a universal character in the source text R. In case of a text composed of the DNA5 alphabet, the "N" character is such a wildcard.

The implication on string matching is the requirement to properly treat the wildcard, for example by denoting each position of a "N" in a match as *N-mismatch*. Figure 2.3 shows an example source text excerpt with an occurrence of a search pattern comprising two *N-mismatches*.

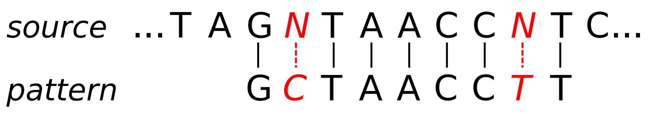


Figure 2.3: *N-mismatch* example showing a source string excerpt and a pattern with two N-mismatches marked red.

2.2.3 Approximate String Matching

Introductions to the field of *approximate string matching (ASM)* have been given by Navarro [99] or Michailidis and Margaritis [91]. Although both works focus on online algorithms, i.e. operating directly on the source text opposed to utilizing index structures for speeding up searches, they offer a good overview.

The main goal of ASM is to find all occurrences of M in R with a predefined upper limit of differences. Thereby different metrics exist to measure the difference between two strings. Two of the most commonly used are the *Hamming-distance* metric and the *Levenshtein-distance* metric.

2.2.3.1 Hamming-distance Metric

The *Hamming-distance* metric originates from the context of error detection and correction on large scale computing machines and was introduced by R.W. Hamming [53]. He describes the comparison of two equal-length strings with the aim of identifying all mismatches, i.e. positions with altered characters. An example is shown in figure 2.4.

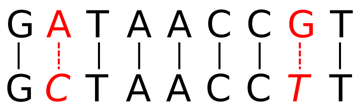


Figure 2.4: *Hamming-distance* metric example with two strings differing in two positions marked red.

A generalization of the problem to find the mismatches between equal-length strings is the *k-mismatch*-problem. The aim is to find all start positions j for the occurrences of M in R with a number $k \geq 0$ of mismatches allowed at maximum. Each of the corresponding substrings of R starting at j has at most k positions with characters differing from M.

2.2.3.2 Levenshtein-distance Metric

The Levenshtein-distance metric, also originating from the field of error detection and correction, is a measure for the differences of two input strings. It was introduced by V.I. Levenshtein [76]. The term edit-distance is often used synonymical, although there are other edit-distance metrics as well.

The Levenshtein-distance metric considers the following character edit operations as a difference:

- `substitute` a character by another
- `insert` a character
- `delete` a character

In order to determine which edit operations have to be applied to transform a string R' into another string M, dynamic programming is utilized. The recursive algorithm shown in figure 2.5 determines the $(m+1) \times (n'+1)$–matrix D of the two strings R' and M with their length n' and m. Each edit operation has a uniform difference value of 1.

$$\begin{aligned}
D_{0,0} &= 0 \\
D_{i,0} &= i \ \forall\, i \in \{1,\dots,m\} \\
D_{0,j} &= j \ \forall\, j \in \{1,\dots,n'\} \\
D_{i,j} &= \min \begin{cases} D_{i-1,j-1} & +0 \quad \text{equal} \\ D_{i-1,j-1} & +1 \quad \text{substitute} \\ D_{i-1,j} & +1 \quad \text{insert} \\ D_{i,j-1} & +1 \quad \text{delete} \end{cases} \\
& \forall\, i \in \{1,\dots,m\} \text{ and } j \in \{1,\dots,n'\}
\end{aligned}$$

Figure 2.5: *Levenshtein-distance* metric recursive algorithm

$D_{m,n'}$ finally contains the Levenshtein- or edit-distance between R' and M.

The positions and types of the edit operations can be obtained by backtracking through the matrix. Figure 2.6 shows an example of two strings differing by a deletion and a substitution. At first the matrix is utilized to determine the total number of edit operations, which is two. The second step is to determine the positions of the edit operations by following the path back through the matrix.

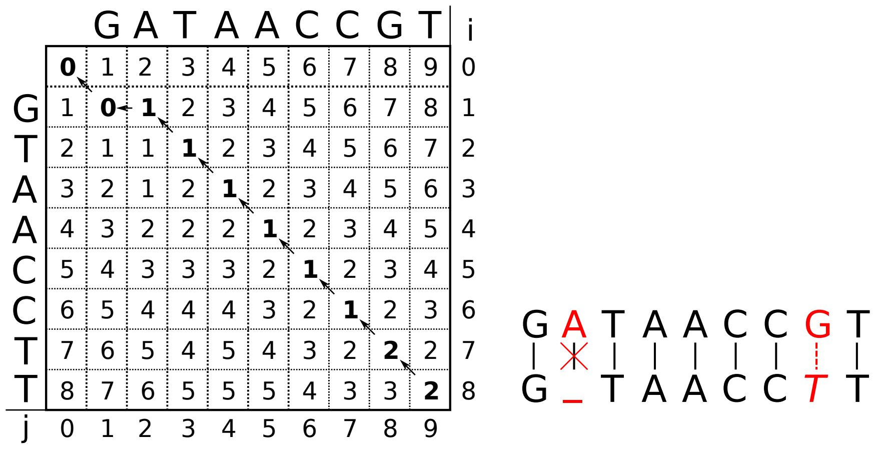


Figure 2.6: *Levenshtein-distance* metric example showing the backtracking matrix as well as the two strings in direct comparison with the differences marked red.

The *k-difference*-problem is the generalization of determining the Levenshtein-distance of two equal or near-equal length strings. The aim is to find all start positions i for the occurrences of M in R with $l \geq 0$ differences allowed at maximum. Each of the corresponding substrings of R starting at i can be obtained from M with at most l edit operations.

2.2.4 Weighted Approximate String Matching

In *weighted approximate string matching (WASM)* the different edit operations are not treated equally with a uniform difference value of 1. Instead individual values are assigned depending on the position as well as the source and target character.

In the Hamming-distance metric each substitution gets a weighted mismatch value depending on the source and target character. The position of a substitution within the search pattern can be taken into account as well, for example by applying a position dependent correction factor.

In the Levenshtein-distance metric, substitutions are treated in the same way as for the Hamming-distance metric. In addition, the edit operations insert and delete get a weighted difference value depending on the character removed or added. The recursive algorithm to determine the dynamic programming matrix can be adopted to the weighted scheme (figure 2.7). The fixed mismatch values are replaced by appropriate weighted values. For each kind of edit operation, the weighted mismatch value is determined by a corresponding function and added to the difference count. These functions return a distance value depending on either both the source and pattern string character ($sub(i,j)$) or the source character only ($ins(j)$ and $del(j)$). More complex approaches could incorporate context information as well.

$$
\begin{aligned}
D_{0,0} &= 0 \\
D_{i,0} &= i \ \forall\, i \in \{1,\ldots,m\} \\
D_{0,j} &= j \ \forall\, j \in \{1,\ldots,n'\} \\
D_{i,j} &= \min \begin{cases} D_{i-1,j-1} & +\,0 & \text{equal} \\ D_{i-1,j-1} & +\,sub(i,j) & \text{substitute} \\ D_{i-1,j} & +\,ins(j) & \text{insert} \\ D_{i,j-1} & +\,del(j) & \text{delete} \end{cases} \\
& \quad \forall\, i \in \{1,\ldots,m\} \text{ and } j \in \{1,\ldots,n'\}
\end{aligned}
$$

Figure 2.7: *Levenshtein-distance* metric weighted recursive algorithm

2.3 Indexing Structures

In computer science indexing structures are widely used to speed up lookup operations in large data sets. An index structure is obtained by preprocessing the source data. In contrast, online algorithms operate directly on it.

For string matching in nucleic acid sequence data, indexing structures have been utilized for a long time and in many applications. Thereby some structures, for example suffix trees and suffix arrays, proofed to be well suited to deal with the special characteristic of sequence data which does not contain any delimiters. Hence a sequence is one large string of characters instead of separated words making for example the widely utilized inverted indexes less suitable.

For this reason, the following sections will introduce several indexing structures focusing on the most relevant ones in the context of this work.

2.3.1 Basic Notations

Consider $R = r_1 r_2 ... r_N$ being a string with $n = |R|$ symbols over the alphabet Σ. A termination symbol $\$$, which is not part of the alphabet, is attached at the end of R.

Each suffix of R beginning at position $i, 1 \leq i \leq n$ can be identified by the term $s_i = R[i, n]$. With this, $s_1 = R$ and $s_n = \$$. Each suffix is uniquely identified by its starting position i in R.

A *generalized index* is an index based on a set of strings $P_1 ... P_M$ which are all incorporated into one index. An often utilized technique to construct an index from these parts is to concatenate all parts to one large string R which is indexed afterwards. The index positions denoting a border between two parts are stored. This is required for string matching because otherwise it would not be possible to identify the artificial substrings which are formed by the concatenation process.

The examples in the follwing sections are based on the nucleic acid sequence string $R =$"GATAACCGT$". When talking about string matching, it is based on the pattern M with the length $m = |M|$.

2.3.2 Suffix Tree

The *suffix tree (ST)* is a well-known index structure which has been of research interest for a very long time. Back in 1968 Morrison presented the PATRICIA tree [93], an early form of the ST. As explicit index it was proposed by Weiner in 1973 [138].

For a string R a ST is a tree with labeled non-empty edges. The labels contain one or more characters, opposed to suffix tries where all edges are labeled with exactly one character. The paths of the tree, i.e. the concatenation of edge labels from the root to a leaf, correspond to the suffixes s_i of R. Their start positions in R, often denoted as occurrences as well, are stored as leaves. Figure 2.8 shows an example of a suffix tree.

A lot of research has been done on efficient construction algorithms resulting in $O(n)$ time and space requirements. A recent review by Barsky and colleagues includes a more

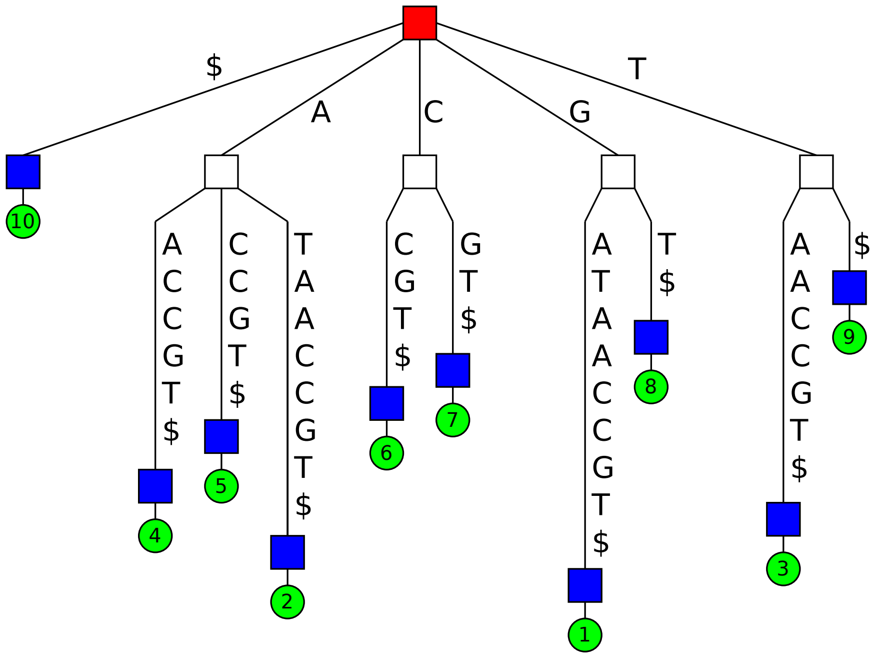


Figure 2.8: Suffix Tree for the example sequence "GATAACCGT$"

detailed description of suffix trees and common construction algorithms [8].

Due to its structure and the suffix position information in the leaves, a suffix tree allows fast searching of substrings contained in R. It takes time linear to the length m of the search string M plus the number of occurrences in R to return all exact appearances ($O(m+occ)$). However, the memory requirements are a major drawback of suffix trees. Unoptimized they require at least $10n$ [8] up to $20n$ or even more memory [1].

To deal with the huge memory demands, several approaches have been proposed. The *generalized kTruncated Suffix Tree (kTST)* [120] and the *ARB-PT-Server* incorporated in the ARB software environment [84] truncate the suffixes in order to reduce the size by limiting the height of the suffix tree. Other approaches, recently reviewed by Barsky and colleagues, utilize secondary storage to circumvent the problem of suffix trees exceeding main memory size for large data sets [8].

2.3.3 Suffix Array

In order to overcome the disadvantage of high memory requirements of suffix trees Manber and Myers proposed the *suffix array (SA)* in 1990 [86]. An overview of suffix arrays was recently given by Grossi [50].

For a string R the corresponding *SA* is the lexicographically sorted array of the suffix start positions. The *SA* space requirements are in the order of $4n$. Exact string matching can be conducted in $O(mlogn)$ time while the construction time is $O(nlog_2|\Sigma|)$ in the best case. Figure 2.9 shows an example of an suffix array.

A disadvantage of bare suffix arrays has been the inability to perform all operations of a suffix tree. Hence Abouelhoda and colleagues proposed the *enhanced suffix array (eSA)* in

index	suffix
4	AACCGT$
5	ACCGT$
2	ATAACCGT$
6	CCGT$
7	CGT$
1	GATAACCGT$
8	GT$
9	T$
3	TAACCGT$

Figure 2.9: Suffix Array for the example DNA sequence "GATAACCGT$"

2004 [1]. It is capable of replacing suffix trees in all applications by adding some additional tables. An *eSA* has a memory requirement of $6n$ and a search complexity for exact matches $O(m + occ)$ like suffix trees.

Furthermore, research has been conducted on *Suffix Arrays on secondary storage* in order to deal with limited main memory sizes. The most recent approach has been proposed by Moffat and colleagues [92]. The space requirements are about $7n$ for nucleic acid sequence data with the search times for exact matches being of the same complexity as for the *eSA*, i.e. $O(m + occ)$.

2.3.4 Compressed Index Structures

In order to further reduce the high memory demands of suffix trees and suffix arrays, compressed index structures were invented creating the field of so-called *full-text self-indexes*. A self-index contains the original source string R in a compressed representation and allows fast substring queries as well.

Self-indexes exist in many variations differing in space and time requirements. In their survey Navarro and Mäkinen give a detailed overview of compressed full-text indexes including a detailed introduction to self-indexes [102].

The first self-index is the *FM-index* invented by Ferragina and Manzini [37]. It is based on the Burrows-Wheeler transformation (BWT) [19] in combination with the data structure of suffix arrays. The BWT permutes the order of characters of the original string by sorting all rotations and taking the last column. This results in a string with clusters of repeated characters if a single character occurs several times in the original string, making the transformed string compressable by for example run-length encoding. Reversing the transformation is possible with only having the transformed string [19]. The FM-index is often named *compressed suffix array (CSA)* as well. With respect to the size of the input data, the search time and space requirements of the FM-index are sublinear.

Another self-index is the *LZ-index* presented by Navarro in 2004 [100]. It is based on the LZ78 compression utilizing a dictionary and replacing repeated occurrences of substrings by pointers to the dictionary [144]. The *LZ-index* has space requirements in order of the original text.

Like enhanced suffix arrays, *compressed suffix arrays* can be enhanced by adding data structures to provide full suffix tree functionality. These enhanced *CSA* are called *compressed suffix trees (CST)* and were first proposed by Grossi and Vitter [51].

2.3.5 Approximate String Matching With Indexing Structures

As online approximate string matching algorithms face the problem of decreasing performance due to increasing text sizes, offline algorithms are of great interest. With the help of pre-processed indexing structures, the search time can be reduced significantly for single queries, although the index generation consumes time as well.

For suffix trees Ukkonen was the first who presented different approaches for approximate string matching [135]. Navarro and colleagues later summarized the three main *ASM* approaches for suffix trees and arrays including a brief introduction [101].

The first approach is *neighborhood generation*. The index is queried for all occurrences of a pattern within the maximum distance k by comparing the pattern to the paths of the tree.

The second approach is *partitioning*. The search pattern is divided into pieces and the index is queried for exact matches of selected pieces. With the original text, the areas around each occurrence of the approximate hit must be verified afterwards.

The third approach, *intermediate partitioning*, is a hybrid between the two latter ones. The index is queried for search pattern pieces allowing fewer errors than for the whole pattern and the occurrences are verified afterwards.

Although in-memory suffix trees can be utilized efficiently for ASM, Barsky and colleagues state that it is still an open challenge for suffix trees on secondary storage [8].

Approximate string matching with the help of compressed indexes has been subject of research, too. In their study Russo and colleagues presented an approach for ASM utilizing the LZ-index and other CSAs/CSTs [116]. They observed high slowdowns even for low error rates.

2.4 Related Software Applications

In the following sections the software applications and frameworks relevant for this thesis are presented. First the relevant database management systems are introduced followed by a presentation of data integration platforms for nucleic acid sequence data. Afterwards the different bioinformatic application programming interfaces (APIs) of interest, targeting sequence analysis, are described. Finally some primer/probe design applications as well as programs of interest to function as a building block for new developments are introduced.

2.4.1 Database Management Systems

A collection of logical associated data is called a *database*. For its electronic management *database management systems (DBMS)* are utilized. They allow fast access and guarantee data integrity, for example by supporting the ACID paradigm (atomicity, consistency, isolation, durability).

Common features offered by many DBMS are handling the concurrent access of several clients which often requires transaction support as well as a query language for defining, querying and manipulating data. Furthermore index structures for fast random access to specific parts of the content, access control, backup and replication support are often provided.

DBMS utilize different models to store data, for example the relational model. It is the base of different widely employed *Relational DataBase Management Systems (RDBMS)* and goes back to proposals for managing large shared databases by E.F. Codd in 1970 [24].

Instead of storing all data using one large record per data entry, the entries are split up and stored distributed over several tables. The relations between the parts are stored by utilizing keys. For data definition, querying and manipulation, many RDBMS support the *Structured Query Language (SQL)*. There are numerous commercial and open source RDBMS, for example *MySQL*, *PostgreSQL* and *SQLite* to name just a few popular open source representatives.

MySQL is an open-source DBMS available for multiple operating systems. It provides various storage engines for different purposes. Some provide special functionality like the in-memory engine and the *Network Data Base (NDB)* engine. The NDB engine is the base of *MySQL Cluster* which allows to install the MySQL DBMS on a computer cluster for fast access and high availability. The *MySQL Cluster* is often used in environments where many reads and rare writes are common [97]. Programmatic access is possible utilizing the MySQL C API or one of the wrappers in other programming languages, for example *MySQL++* for C++ [98].

Another popular open-source RDBMS is *PostgreSQL* [112]. It is available for Linux, UNIX derivates and Windows offering native programming interfaces for various programming languages. For C++, the library *pqxx* is available working on top of the libpq C API [80].

SQLite is a C/C++-library offering a standalone SQL database engine [123]. It does not provide a server architecture and can be seen more as a management library for flat file databases. For different programming languages wrappers for the C API exist.

Other common data models employed in DBMS are the object oriented model and the hierarchical model. Hierarchical DBMS utilize parent/child relationship to manage data. An example is the *ARBDB* which is part of the ARB software environment [84] (refer to section 2.4.4.2). The *ARBDB* offers its own C/C++ API including transactions and different query functionality, although concurrent access by different users is not supported.

2.4.2 Nucleic Acid Data Integration And Access

Various applications and frameworks offer different ways to search and retrieve nucleic acid sequence data from different repositories or to access and integrate nucleic acid data into new applications. The following sections will introduce different existing approaches ranging from web-oriented remote database access, data warehouses to integration platforms.

2.4.2.1 Remote Sequence Database Access

The EMBL EBI, provider of the *EnsEMBL* and *EnsEMBL genomes* databases, offers several remote APIs to access their databases programmatically. The *dbfetch* REpresentational State Transfer (REST) API enables the retrieval of entries by their respective identifier or a unique accession number [27]. The output format depends on the content type of the database. For nucleic acid sequence data, the *FASTA* and the *EMBL Bank XML* file formats are supported. The number of concurrently returned entries is limited to 200 per query. Furthermore, with *WSDbfetch* a web service Simple Object Access Protocol (SOAP) API is provided with the same functionality as *dbfetch* [139]. Additionally several EMBL EBI databases hosted on MySQL servers are directly accessible utilizing a MySQL API [35]. Finally the databases are available for download as flat file exports and MySQL database dumps.

The NCBI offers similar access to their databases. The *Entrez Programming Utilities* provide a REST API and a Web Service API to access *GenBank* as well as various other databases [104]. The database entries are returned as text files. Different formats like for example *FASTA*, GenBank flat file and different XML formats are supported.

2.4.2.2 Biological Data Warehouses

One category of data integration platforms are *data warehouses*. They integrate data from different sources into a unified schema. This often requires the transformation of the data representation. The goal of a data warehouse is the support of data analysis by online analytical processing (OLAP) or data mining, i.e. extracting new patterns and relations.

The existing data warehouses for biological data integrate databases from different fields of biology, for example nucleic acid sequence and protein data. Their main goal is to support researchers without deep programming knowledge in performing large scale analysis. This is achieved by providing predefined tools and workflows accessible over web-interfaces or a web-service API.

One well established and representative biological oriented data warehouse is *BioMart* provided by EMBL EBI [122]. *BioMart* offers public server access besides web services and native Java APIs [35]. It has been recently extended to function as a data federation framework for locally distributed databases [143]. It provides a central portal [52] and recently integrated the databases from the *Ensembl Genomes* project as well [69].

2.4.2.3 Biological Data Integration Platforms

A different type of integration platform besides data warehouses are integrated frameworks, for example the *Galaxy toolkit* for biomedical genome analysis [43]. It is web-based and aims at providing simplified mechanisms for coupling external data resources with the available analysis tools. Further enhancements have been developed to cope with the ongoing fragmentation and specialization of data sources [16].

Another integration platform is the *ACNUC* biological sequence database system providing remote access to the database collection at the Pole Bio-Informatique Lyonnais (PBIL) [47]. Some of the major sequence databases like EMBL or GenBank are cloned and integrated. For data selection *ACNUC* provides its own query language. Access is provided by a range of APIs in different programming languages as well as by a browser interface. Furthermore ready-to-use programs are provided.

The *Sequence Retrieval System (SRS)* [36] started as an integration platform for flat file collections providing a unified text based search interface. Currently *SRS* provides an integrated platform for biological data search and retrieval. It currently integrates a wide range of biological data sources such as databases for protein and nucleic acid sequence data, protein interactions as well as content generated by studies and other analysis to name some. In the meantime it evolved to a commercial product distributed by BioWisdom Ltd. [15].

The data contained by a *SRS* server can be queried and accessed utilizing *wgetz*, a representational state transfer (REST) API. For public available SRS-servers, the REST API can be accessed freely. Other APIs exist, but are only available commercially, for example a web service SOAP API.

Using the REST API, the *SRS* system is queried by building a URL providing all required information comprising the databases to query, the field identifiers and the values to search for. Some further parameters allow to influence the returned data, for example to get the matching values in a certain text file format like *FASTA* or *EMBL Bank* (refer to section 2.1.4) instead of a HTML-page.

Different approaches exist to provide wrappers for the SRS-APIs allowing the programmatic access with different programming languages. *SRS.php* targets web-applications and provides a wrapper around the *SRS* SOAP API [7]. Another unpublished approach is the *SrsUrlApi* written in Perl [109]. It provides a client library to access the *SRS* REST API.

2.4.3 Bioinformatics Application Programming Interfaces

In the field of sequence analysis and data retrieval many different application programming interfaces (APIs) in different programming languages exist. A brief overview is given by Stajich and Lapp [127].

Some of the most prominent APIs are federated in the Open Bioinformatics Foundation (OBF) [106], namely *BioJava* [56], *BioPerl* [126], *BioPython* [23] and *BioRuby* [46]. These are also called the *Bio** projects [87].
All *Bio** projects are open-source frameworks providing capabilities for nucleic acid and protein sequence analysis, manipulation and alignment to name a few. Data access is provided by routines for loading and storing the common flat file formats (refer to section 2.1.4). One common achievement is the *BioSQL* database schema (refer to section 2.1.3). It is part of the *Open Biological Database Access (ODBA)* initiative which has the defined goal to establish a generic and standardized way of accessing biological data sources [105]. As of now, the current status and further development of the *OBDA* project is unclear as the last update was back in 2002 [105]. Furthermore at least *BioRuby* has capabilities to access remote online resources programmatic, for example the online databases of the three large primary sequence data providers (refer to section 2.1.5).

Bio++ is a set of object oriented C++ libraries offering APIs for sequence analysis, phylogenetics, molecular evolution and population genetics [31]. Data can be loaded and stored from and to different flat file formats including *FASTA* and GenBank files. Access to primary sequence databases is granted indirectly by an access module for the ACNUC biological sequence database system [47] (refer to section 2.4.2.3).

Seqan is a C++ template library which provides various generic and efficient algorithms and data structures for sequence analysis [30, 44]. It offers flat file I/O capabilities as well as several search index implementations like enhanced suffix arrays and suffix trees including those on secondary storage. Furthermore, some online approximate string matching algorithms have been implemented.

The *NCBI C++ Toolkit* provides both general purpose and biotech-related libraries [103, 136]. On the general purpose side there is the DBAPI. It provides a low level API which abstracts from vendor specific RDBMS APIs. With this API databases can be accessed in an object oriented manner and queried by SQL queries.
Some features related to bioinformatics and sequence analysis are the object manager facilitating the access to sequence data. It is responsible for managing the details of loading data from heterogeneous data sources. The available data loading capabilities are focused on the NCBI GenBank database and different flat file formats.

2.4.4 Primer/Probe Design Applications

The next sections will first present the main goals in primer/probe design and evaluation. Afterwards dedicated primer/probe design applications are presented followed by short read mappers which may be of interest as starting point for developing new approaches.

2.4.4.1 Primer/Probe Design And Evaluation

Primer and probes are of interest for various biological applications including microbial diagnostics (refer to section 2.1.2).

In nucleic acid sequence based primer/probe design, several parameters are common. The aim of them is to predict certain wet-lab behavior to exclude unsuited candidates prior to in-silico and wet-lab evaluation. The first commonly employed parameter is the number of “G” and “C” bases a candidate stretch is allowed to contain. This is often referred to as *GC-content* and abbreviated *G+C*. The second metric often utilized is a prediction of the melting temperature based on the bases contained in a candidate stretch, denoted as *melting temperature*. An often utilized simple rule to obtain an estimation is the Wallace rule [137]:

$$T_d = 2°C(A + T) + 4°C(G + C)$$

The number of “A” and “T” times two plus the number of “G” and “C” times four is the estimated melting temperature in degree celcius.

In order to evaluate the *sensitivity* and *specificity* of a primer/probe (refer to section 2.1.2), it is important to get all hits within a defined distance to all sequences incorporated in the analyzed data collection. Only with a comprehensive list of potential binding sites the identification of potentially cross-reacting non-target sequences (and respective organisms) is possible.

2.4.4.2 ARB Software Environment

The *ARB* is a software environment for storing and processing sequence data [84]. It started as a software for ribosomal RNA data but can be utilized for DNA or protein data as well. The ARB comprises a graphical user interface as well as command line applications.

Its first central component is the *ARB database (ARBDB)*, a proprietary hierarchical in-memory database including a text based search interface. It supports storing sequence data and its annotations along with processed and descriptive data like clustering information in form of phylogenetic trees in one database. Various importer and exporter for different sequence flat file formats are available.

Another central component is the *ARB PT-Server*, a search index server based on a truncated suffix tree fully incorporated into the software environment. Relying on an ARB database as the only valid input source, it is capable of constructing an index comprising all nucleic acid sequence entries contained. Ambiguous base characters are all treated as “N”. To be fully functional, the *PT-Server* requires the corresponding database being available during application.

Based on the fast query capabilities of the PT-Server, ARB offers software tools for in-silico oligonucleotide primer and probe design and evaluation.

The application *ARB ProbeMatch* is capable of identifying all occurrences of a pattern within the different sequence entries comprised in a database. Utilizing the Hamming-distance metric, *ProbeMatch* is capable of carrying out approximate as well as weighted approximate string matching with a user-defined maximum number of mismatches allowed. A list of the matches is returned containing detailed match information. For identifying the sequence entry hit its database identifier as well as its name are included. Furthermore, for quality assessment, the number of mismatches, the number of N-mismatches, the weighted mismatch value as well as position information are returned. The position information takes alignment reference entry information into account if available. Finally match context information is returned. It is composed out of the match sequence context at nine positions at 3′ and 5′ ends and a differential alignment representing equal positions, substituted bases and N-mismatches.

The *ARB ProbeDesign* application utilizes the *ARB PT-Server* to search potential primer and probe target sites for a user selected group of sequence entries. These entries can be marked by either utilizing the ARB phylogenetic tree viewer, manually selecting sequence entries in a database entry list view or by utilizing the database search interface. Prior to running the design process, several parameters can be defined comprising the targeted length, thermodynamic criteria in form of minimum and maximum melting temperature and the signature characteristic minimum and maximum GC-content. Further parameters are the minimum group coverage and the maximum number of outgroup hits allowed. After the design process finished, a list with signature candidates is returned. Each entry of the list is composed of the signature candidate, the target site, the thermodynamic properties as well as a quality assessment.

2.4.4.3 Further Approaches

Another recent approach for primer/probe design is *CMD/PSID*. It targets the fast local on-the-fly computation of unique signatures targeting a single sequence in a set of DNA sequences [74]. The algorithm relies on the Hamming-distance metric and is not capable of taking ambiguous base characters into account.

In contrast, *Insignia* is an online platform capable of comprehensively identifying signatures for gene or genome nucleotide sequence collections [110, 111]. Suffix tree supported pre-processed comparisons between the sequences are stored in a database and utilized for signature computation.

Another approach is *CaSSiS* which pre-computes comprehensive collections of signatures for a given set of nucleic acid gene or genome sequences in conjunction with clustering information like phylogenetic trees [6]. It is based on the *ARB PT-Server* (refer to section 2.4.4.2) for fast approximate oligonucleotide string matching enabling the calculation of perfect signatures, i.e. matching the target sequence or group perfect. If no such signature exists, candidates with maximal group coverage and minimal out-group hits can be determined as well.

2.4.4.4 Short Read Mapper

Besides primer and probe design, another field of bioinformatics leveraging index structures is the assembly of genomes from short reads produced by next-generation sequencers.

For each short read the best matching position compared to a reference genome sequence is determined. To accomplish this task, it is sufficient to find one or a few positions where the edit-distance of the pattern (short read) is minimal. With next-generation sequencers being capable of producing millions of reads a day [90], the assembly requires many queries making it a very compute intensive task. In their survey, Li and Homer recently presented current developments and existing tools for short read mapping [79].

One notable short read mapper is *BWA* [77]. To speed up the mapping process, an FM-index of the reference genome is utilized (refer to section 2.3.4). The index is queried non-heuristically by sampling short substrings of the reference and comparing the query pattern to them allowing few differences. Depending on the settings, it is capable of returning all exact or approximate matches of a short read. Default is to return only the position with the lowest difference. Besides the position within the reference genome, some auxiliary information is provided. This includes the hit position and a CIGAR string formatted output of the target site (refer to section 2.1.2).

2.5 Computer Hardware Architecture

The specific hardware architecture influences the development of an algorithm or program. The latter must take care of specific constrains and limitations of the hardware in order to perform well.

The next sections will give a brief overview of the memory hierarchy of common mainstream hardware architectures followed by an introduction to parallel architectures of interest within this thesis.

2.5.1 Memory Hierarchy

Development of computer memory and storage faces rivaling aims: memory should be as fast as possible while being as big as possible. Up to now no technology has been able to fulfill both goals at once at affordable production costs. Memory tends to be either fast, small and expensive or slow, huge and cheap.

Modern microprocessors require fast access to data items in main memory to avoid starving of computations. Computations of the central processing unit (CPU) are carried out on internal *registers*, although not all data fits into these as they are very small in the order of tens of bits depending on the specific architecture and purpose. For example the x86-64 architecture provides 64-bit general purpose registers. Register access is often possible in 1 CPU cycle, i.e. less than a nanosecond for common clock rates in the range of 1 to more than 3GHz.

The access time to *main memory* is rather high compared to registers. In order to soften this disadvantage, a *cache memory hierarchy* is utilized, often up to three levels. Caches are used to reduce the average time to access a data item in memory by storing copies of the data and providing fast access in the order of a few to tens of CPU cycles, i.e. in a few nanoseconds. Cache sizes and the number of levels vary from processor to processor, but some key parameters are fairly the same. The first cache level provides the fastest access, often divided into data and instruction cache, and has sizes in the order of kilobytes. The second level is a little bit slower. It is often for all kinds of data and in the size of hundreds of kilobytes. Some processors incorporate a third level of cache which can be in the size of megabytes.

The *main memory* access time is slower compared to the caches and lies in the range of 10s of nanoseconds, although the size is significantly larger. Main memory capacity can reach up to several 10s of GB. It is referred to as primary storage as well and provides non-persistent data storage. The CPU can access main memory directly over either a bus system or utilizing point-to-point connections. The latter approach has become standard and is available under different names, for example HyperTransport by AMD or QuickPath Interconnect (QPI) by Intel. The memory transfer rates reach tens of GB per second in state of the art processors.

The next level of memory is the *secondary storage*, sometimes called *external memory* as well. It is still dominated by *magnetic disk* drives providing huge amounts of persistent storage. Secondary storage can usually not be accessed by the CPU directly. I/O channels manage the access and transfer of data to and from main memory. Before transferring data from a magnetic disk drive to main memory, the so called seeking is required, i.e. the re-positioning of the read/write-head to the correct position on the disk. Afterwards the required blocks of data can be transferred. If they are not on consecutive positions on the disk, a new seek may be required in between. This slows down the transfer for random access pattern significantly. If the data is transferred from consecutive blocks, the re-seek is avoided which allows higher transfer rates. Therefore the time to access data on a magnetic drive is the sum of seeking and transfer times. The access time is in the range of milliseconds, which is extremely slower compared to main memory. Regarding the size, common magnetic disks have reached capacities of up to 3 TB and offer read transfer rates of about 150 MB per second.

Recently the so called *solid-state drives (SSD)* have become a fast and affordable alternative to traditional magnetic drives. SSDs have no movable parts, provide better random I/O and in general higher transfer rates. They are build out of non-volatile flash-memory. Due to mainly the price and power consumption, the average sizes of SSDs are smaller than magnetic disks, although they have outreached the TB border as well. The access time is in the range of 100s of microseconds and the transfer rates outreached 500 MB per second.

Figure 2.10 shows a simplified overview of the nowadays common memory hierarchy divided into primary and secondary storage. It also shows the order of the nowadays common sizes. Further levels like tertiary storage, for example tape drives utilized for backups, are omitted.

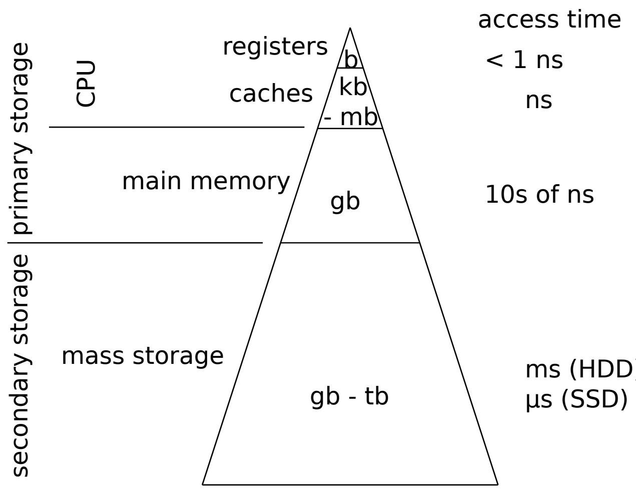


Figure 2.10: Simplified memory hierarchy overview

2.5.2 Parallel Computer Architectures

In the past the performance of microprocessors has been increased by raising the clock rate and optimizing the circuit design. Unfortunately, increasing the clock rate is limited by the fact that the power consumption goes up exponentially at the same time. With a higher power consumption the heat dissipation rises as well, requiring higher cooling efforts. Although this trend can be softened by shrinking the processors integrated circuits, a limit has been reached around 4 GHz where it became economical and ecological problematic to further continue this way to increase microprocessor performance. The *power wall* has been reached.

Luckily Moore's law is still valid. It states that the number of transistors in a microprocessor doubles every 18 months. This increase can still be utilized for improving microprocessor performance by design. Instead of a single core, around the year 2005 the chip developers started to design mainstream microprocessors with multiple processing cores, so-called *multi-core processors* or single *chip multiprocessors (CMP)*. They combine either several separate single core dies in one processors or several processor cores on one die. Up to now, this design has become common for desktop computers and server systems.

Systems with two or more processors sharing the same global main memory and (physical) address space are denoted *shared memory* systems. All multi-core processor systems fall under this category. Formerly shared memory systems have been mainly *multi-processor* systems available in high performance computing (HPC) environments and for enterprise servers.

Shared memory systems are further sub-classified into systems with *uniform memory access (UMA)* and *non-uniform memory access (NUMA)*. For *UMA* systems all processors are connected uniformly to the physical memory via a shared bus. Hence they have the same access latency and share the memory bandwidth. Within *NUMA* systems, the latency to access memory depends on its physical location relative to a processor. As most new microprocessors utilize point-to-point connections to access main memory, for example Intels QPI or AMDs HyperTransport, current multi-core processors and with this the majority of newly sold computer systems are *NUMA* systems.

In contrast to *shared memory* systems, in *distributed memory* systems each processor has its own private memory. Each processor runs its computations utilizing its local memory. If a processor requires access to remote data, it is necessary to communicate with one or more remote processors. Data needs to be transferred via a connection system, for example common ethernet network or InfiniBand communication links. In HPC *distributed memory* systems are common. With the advent of multi-core processors, many of the newer cluster systems are a hybrid between *distributed memory* and *shared memory*. This must not be mixed up with *distributed shared memory* systems where each node has access to the whole local and remote memory via a shared global address space.

2.6 Program Optimization And Parallelization

Optimization and parallelization plays a huge role to improve memory requirements as well as the runtime performance of computer programs.

The following sections will present techniques for optimization and parallelization starting with compact representations of nucleic acid sequence data. Afterwards general purpose encoding strategies utilized for data compression are introduced. Finally a brief overview of common parallel programming frameworks and APIs is given including an introduction to the different sources of parallelism.

2.6.1 Nucleic Acid Sequence Compact Representation

Compact representations are utilized to reduce the amount of memory required to store nucleic acid sequences.

The *DNA4* alphabet with its four symbols can be represented utilizing two bits per base character allowing four bases per byte. With this it is possible to store 16 base characters in a 32 bit word or 32 base characters in a 64 bit word. Figure 2.11 shows a possible encoding and a sample byte.

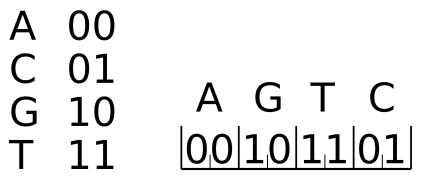


Figure 2.11: *DNA4* alphabet encoding with a sample byte

For the *DNA5* alphabet the representation with four bases per byte can not be applied. In order to provide a compact representation requiring a low number of bits per base, the five different characters are mapped to an integer representation first: $\mathrm{val}(\text{"N"}) = 0,\ \mathrm{val}(\text{"A"}) = 1,\ \mathrm{val}(\text{"C"}) = 2,\ \mathrm{val}(\text{"G"}) = 3,\ \mathrm{val}(\text{"T"}) = \mathrm{val}(\text{"U"}) = 4$. A sequence can now be stored as the sum of its base characters codes to the base of five:

$$pval = \sum_{i=1}^{n} 5^{n-i} * val(S_i)$$

This representation allows to store a sequence of 13 bases in a 32-bit integer value (as $5^{13} < 2^{32}$) or 27 bases in a 64-bit integer value (as $5^{27} < 2^{64}$). In both cases one base character utilizes $\frac{\log 5}{\log 2} \approx 2.3$ bits. For 32-bit as well as 64-bit integers there is at least one bit unused by this representation. It can be utilized as a *stop bit*. This allows to optionally encode and decode the length of the compact sequence implicitly, i.e. without a separately stored value. The stop bit enables the storing of leading "N"s which would otherwise get lost when decoding.

In figure 2.12 the encoding for the sample sequence "AGNTC" including the stop bit is presented. The position of the stop bit is calculated by solving the inequation $5^{length} < 2^x$. For example for a length of 5, the inequation $5^5 < 2^x \Rightarrow \log_2 3125 < x$ results in a stop bit position of $x = 12$.

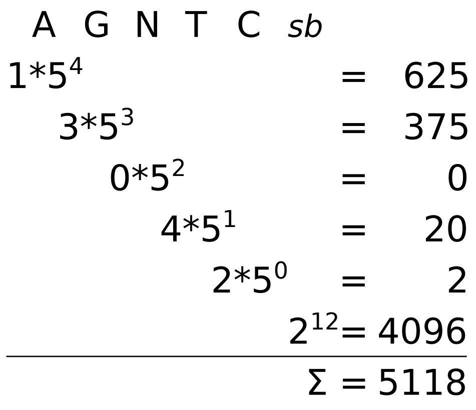


Figure 2.12: *DNA5* compact representation encoding example

To decode the sequence it is necessary to detect and remove the stop bit first. For each length it is possible to determine the stop bit position as stated above. To decode an 64-bit integer the iterative length detection checks the compact sequence value for being lower than the integer value if only the stop bit is set. If true, this process is repeated for the next smaller length until the compact sequence value is greater or equal. For the compact representation of the example sequence "AGNTC" this will be the case for $5118 \geq 2^{12}$ with 12 being the corresponding stop bit position for the length 5.

The stop bit can now be removed and the sequence can be decoded applying modulo arithmetics. Figure 2.13 demonstrates the stop bit removal and decoding for the example sequence "AGNTC".

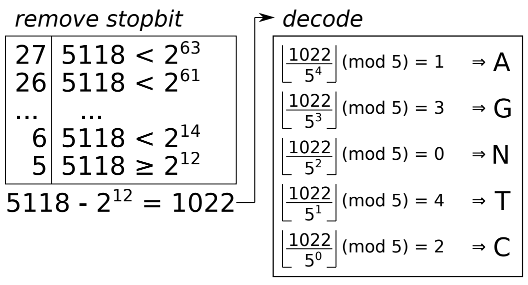

remove stopbit
27 5118 < 2^63
26 5118 < 2^61
... ...
6 5118 < 2^14
5 5118 ≥ 2^12
5118 - 2^12 = 1022
decode
⌊1022/5^4⌋ (mod 5) = 1 ⇒ A
⌊1022/5^3⌋ (mod 5) = 3 ⇒ G
⌊1022/5^2⌋ (mod 5) = 0 ⇒ N
⌊1022/5^1⌋ (mod 5) = 4 ⇒ T
⌊1022/5^0⌋ (mod 5) = 2 ⇒ C

Figure 2.13: *DNA5* compact representation decoding example

If the sequence to compress is aligned (refer to section 2.1.2), i.e. it contains dot (".") and hyphen ("-") symbols in addition to the *DNA5* alphabet, a simpler compact representation scheme can be utilized. Each *DNA5* symbol is stored using 3 bits. As dots and hyphens normally occur in longer stretches, run-length encoding is utilized. If a dot or hyphen occurs, it is stored in an integer representation followed by the number of occurrences. This representation can not be used with single integer values as storage. It is meant for usage in arbitrarily long sequences of consecutive bytes.

2.6.2 Optimized Data Representation

Different techniques exist to reduce the amount of memory required by certain data. They are often utilized in data compression, for example.

Prefix codes can be applied to provide variable length payload information after a corresponding prefix. All prefixes must conform to the requirement demanding no prefix being prefix of another one, also known as the Fano condition. A simple example would be a stream of bits which contains a mix of 31 bit and 63 bit payload items. For this two prefixes are sufficient, so "0" and "1" can be taken. For example "1" indicates that the payload is 31 bit wide, "0" indicates a 63 bit payload. Without the prefix, all payloads would require the size of the largest payload, i.e. 63 bit in the example. With prefix codes the memory consumption for all 31 bit payload items can be reduced.

Huffman encoding originates from information theory and is used to compress data lossless according to its entropy [58]. A simple example would be a tree with labeled edges. For the labels, a variable length code table is generated. In doing so, more frequently occurring edges are represented by a shorter code. If the tree is now for example transferred into a flat representation for storing and transferring it, the average output size gets smaller by replacing the edge labels with their corresponding codes.

Delta encoding is a compression technique where the difference between two consecutive values in an array is calculated instead of storing the full value each time. For example, a list of integer values 170200, 170385, 170544 etc. is delta encoded as 170200, 185, 159.

The *bit field idiom* allows to store multiple logical values in a compact representation. For trees, it is utilized by so called *branch masks* where each bit corresponds to an outgoing edge. If a bit is set, the edge is valid. This allows to store only the valid outgoing references reducing the average amount of memory required for sparse trees. For example taking a suffix tree and an alphabet size $|\Sigma|$, i.e. the width of the branch mask as well, bit i corresponds to character i of the alphabet. For *DNA5* this results in a mask of five bits. If there is no downward edge beginning with a base i, the reference is omitted. An additional benefit of the branch mask in case of the labeled suffix tree edges is that the first character of every downward edge is already defined implicitly.

2.6.3 Parallel Programming

As multi-core processors are common nowadays and high performance computing is utilized a lot in different scientific and commercial fields, parallel programming for speeding up computations is in the focus of many programmers. Several existing frameworks and APIs based on different paradigms try to make parallel programming more convenient.

The next section will introduce the common sources of parallelism. Afterwards, some widely employed programming frameworks and APIs targeting the different memory architectures, i.e. distributed or shared memory, are presented. The focus lies on approaches for the programming languages C and C++.

2.6.3.1 Sources Of Parallelism

There are different sources of parallelism in computer programs. Most often three types are distinguished, *instruction level*, *data level* and *task level parallelism*. In order to utilize the hardware efficiently, these must be identified and mapped optimally to the available hardware resources either automatically, for example by the compiler, or manually by the programmer utilizing parallel programming frameworks and APIs.

The *instruction level parallelism (ILP)* is commonly utilized without requiring the programmer to take special care. The compiler provides optimized code which is processed by the microprocessor. Each instruction is decomposed into instruction fetch, instruction decode, execute, memory access and write back. Although the actual decomposition depends on the specific processor architecture and may be more fine grained. The distinct instruction parts not requiring the same functional unit of the processor can be executed parallel in a pipelined fashion. In case of superscalar processor designs, some of the functional units are available multiple times to allow parallel execution of the same instruction parts up to a certain degree.

Another source for concurrent execution is the *data level parallelism (DLP)* also known as loop-level parallelism as it can be found in program loops. It corresponds to the *single instruction, multiple data streams (SIMD)* classification according to Flynn's taxonomy for computer architectures [38]. The available data to process is split up and each participating computational resource processes one or more chunks of data. To each chunk the same operation is applied. One form of *DLP* is called *embarrassing parallelism*. Here the data items and computations on them are completely independent. No synchronization is required for parallel execution which makes the distribution among the available resources easier.

Finally, the *task level parallelism (TLP)* corresponds to the *multiple instructions, multiple data streams (MIMD)* classification within Flynn's taxonomy [38]. *TLP* can be either of the form *single program, multiple data (SPMD)* or *multiple program, multiple data (MPMD)*. For *SPMD*, the same program is executed for different data items, although in contrast to *SIMD*, the execution is not performed in lockstep. *MPMD* is the form where different computations are performed on different data items. An example for a programming strategy utilizing *MPMD* is the master/worker approach.

2.6.3.2 Parallel Programming With Threads

For shared memory systems, C/C++ programs can rely on different threading library implementations, for example *pthreads* on Linux or the *winAPI threads* under Windows. One example for an implementation abstracting from the platform specific ones is the *boost::thread* library [18]. Thread libraries provide programming capabilities to leverage data level or task level parallelism alike.

Threading libraries require the programmer to take care of thread creation, thread destruction and synchronization as all threads share the same address space.

To avoid certain thread management tasks, *threadpools* are utilized to reduce the creation and destruction overhead. Threads are created once and can be requested from the pool. If all threads are occupied, a requester must wait till a thread has completed its work. The destruction of the threads is issued by the threadpool. An implementation of a threadpool for *boost::threads* is provided by *boost::threadpool*.

Thread synchronization requires the programmer to properly utilize *locks* which allow sequential access to certain program parts, i.e. only one thread executing it at a time. When utilizing different locks at once, deadlocks may occur. Locks are provided as part of the *boost::thread* library. In order to avoid waiting times, the access to shared data structures should be reduced to the minimum by locking at the latest possible moment. In order to provide proper thread-safe interfaces without so-called self-deadlocks, an important rule is to avoid calling a public accessible method from any method of the same class.

Another way to synchronize access to shared memory are atomic data structures, short *atomics*. Atomics can be utilized for simple data types like integers and are based on the atomic *compare-and-swap (CAS)* CPU instruction. Before writing a value to memory, CAS compares it to the previous value read before the computation to check if it has been altered in the meantime. If not, the value can be written safely, if yes, the computation has to be repeated with the new value until it succeeds. The *boost::atomic* library provides the above described functionality for the C++ build-in data types.

A different approach would be to avoid synchronization by replicating resources if possible [2]. With this, locks and atomics can be avoided, although higher memory demands are a drawback besides further programming effort to recombine computational results.

2.6.3.3 Message Passing Interface (MPI)

The Message-Passing Interface (MPI) has been designed by the members of the MPI Forum [95], comprising parallel computer vendors and software developers. It offers a portable message-passing system supporting the development of scalable parallel applications. Its main target are distributed memory architectures, although it can be utilized on shared memory systems as well.

Besides commercial implementations, there are different open source implementations of the MPI standard for the programming language C, for example *OpenMPI* [48] and *MPICH* [49] in its current version *MPICH2* [94]. They can be utilized in C++ code as well, either directly or by employing an object oriented wrapper like the *boost::MPI* implementation [18].

The MPI standard currently comprises two parts, the MPI-1 and the MPI-2 functionality. The MPI-1 standard covers the essential interface for a message-passing system. It can be divided mainly into four parts: *communicators*, *point-to-point* communication, *collective* communication and *derived datatypes*. MPI-2 offers extended functionality like parallel I/O and remote memory operations [95].

Within MPI, all communication between the participating processes is done via messages. The messages are transferred utilizing so called *communicator objects*. The default communicator `MPI_COMM_WORLD` is initialized during MPI startup. All participating processes are assigned a unique identifier within each communicator called *rank*. The process with rank 0 is usually the master process taking care of initialization and controlling of computations, although it is possible to construct hierarchical controlling structures at runtime as well.

The communication is done by either *point-to-point* or *collective* operations. Point-to-point operations can be performed either blocking or non-blocking. For each option a set of functions is available. When utilizing the blocking `MPI_Send()`, the specified receiver must receive the message with `MPI_Receive()`. After the functions returned on both sides, the transfer is terminated. The equivalent non-blocking functions `MPI_Isend()` and `MPI_Ireceive()` return immediately. The receiver must perform an extra operation `MPI_Wait()` to check whether the data has been transferred.

Besides the *point-to-point* operations where only two processes participate in a data transfer, the *collective* operations allow *1-to-N*, *N-to-1* or *N-to-N* communication.

The function `MPI_Bcast()` allows one sender to broadcast a message to a group of receivers (*1-to-N*). Sender and receiver call the same function with appropriate parameters. In contrast, the `MPI_Scatter()` function (*1-to-N*) allows the distribution of a data vector to a group of processes. A version which allows customization of the distribution scheme is the `MPI_Scatterv()` function. The data distributed can be gathered by either `MPI_Gather()` respectively `MPI_Gatherv()` (both *N-to-1*) to re-build the partitioned vector, or `MPI_Reduce()` to reduce the gathered data into a smaller vector by applying defined operations.

One important *N-to-N* operation is the `MPI_Barrier()` which is utilized to synchronize processes without exchanging data. Each process will continue computations only after all participating processes have reached the barrier. Further operations for *N-to-N* data transfer are `MPI_Alltoall()`,`MPI_Allgather()` and `MPI_Allreduce()`.

The data to transfer can be either of a standard or a user defined data type. For each standard type in C/C++ there is a MPI equivalent datatype like `MPI_INT`, `MPI_DOUBLE` and `MPI_CHAR` to name some. Furthermore it is possible to define customized data types using the `MPI_Type_*` function, for example for C structures. For all data types transfer-buffer management is in the responsibility of the programmer, for example the size of the buffer on the receiving process must be sufficient.

2.6.3.4 Further Approaches

As multi-core processors have become common, various frameworks and APIs are available trying to ease the efforts of shared memory parallel programming.

Open Multi-Processing (OpenMP) is an API targeting shared multiprocessing with C/C++ or Fortran programs on various platforms, i.e. different microprocessor architectures and operating systems including Linux and Windows [108]. It relies on multi-threading where certain tasks are distributed among a number of working threads, and takes care of the thread management. In order to execute certain parts of the code in parallel, i.e. loops or nested loops, they need to be marked. The compiler translates the marked regions into parallel code and the runtime engine takes care of the execution. Only the marked parts will be executed parallel while the rest of the program runs sequential. *OpenMP* is capable of dealing with data as well as task level parallelism.

Intel Threading Building Blocks (Intel TBB) is a C++ template library providing data structures and algorithms which allow operations to be treated as tasks [61]. It does not require a programmer to take care of thread-management in contrast to the thread-implementations described in section 2.6.3.2. A runtime engine takes care of the distribution of the tasks to the available cores of a multi-core processor.

Two approaches targeting not only multi-core processors, but for example graphics processing units (GPUs) as well, are NVIDIA *Compute Unified Device Architecture (CUDA)* [26] or the *Open Computing Language (OpenCL)* framework [107]. Both offer their own C-based programming language to write computing kernels. Initially the main goal was to provide a way to utilize graphics cards for general purpose computations. This goal shifted towards providing a generic way to access different kinds of resources with one common interface. Due to its origins, *CUDA* and *OpenCL* are more targeted towards data level parallelism.

2.6.4 Parallel Index Construction Approaches

As the different index structures described in section 2.3 are crucial building blocks in several fields of computer science in general as well as for bioinformatics aided nucleic acid sequence analysis, fast index construction is of great interest.

Research in the field of parallel construction of index structures led to numerous approaches. For parallelization of suffix tree construction these approaches target either shared memory multi-core systems [134], cluster computer systems [42, 88, 141] or grid environments [22]. On distributed memory systems MPI is utilized to speed up construction for large input sequences.

Alternatively, Lee and colleagues focused on transforming suffix arrays into suffix trees utilizing MPI [75] which can be of interest in conjunction with the MPI driven build of suffix arrays presented by Kulla and Sanders [71].

For self-index structures, Zhang and colleagues presented a parallel FM-index construction algorithm on shared memory multi-core systems [142].

3 Results: Unified Data Access Framework, PTPan Index Structure And Applications

The following sections will give an overview over the major results of this thesis. First the *Unified Molecular Data Access (UMDA)* framework concepts are presented. Afterwards the *PTPan* index is described comprising the structures and construction algorithm. This is followed by the *PTPan* based applications supporting molecular diagnostics as well as the conducted optimization and parallelization efforts. Finally insights on the software components developed will be given, including the integrated system approach, i.e. the *UMDA Primer/Probe (UPP) Designer*.

3.1 Unified Molecular Data Access Framework

3.1.1 Motivation

As presented in section 2.1.3 various different genome database schemas exist. Different projects rely on them to provide their databases in a structured way (refer to section 2.1.5). Because of this heterogeneity of data sources, Jagadish and Olken demand a standardized interface [63]. There are different approaches to allow access to different sequence data sources.

Various data warehouses and integration platforms (refer to sections 2.4.2.2 and 2.4.2.3) are aimed at providing a unified view on the data from numerous source databases. The integration platforms offer data mining capabilities or read-only access over web-frontends to non-programmer users as stated by Töpel and colleagues [133]. Some approaches provide an application programming interface (API) as well.

In addition various application programming interfaces offer sequence data access and analysis capabilities, for example the *Bio** projects to the *BioSQL* database or *Bio++* to the *ACNUC* data integration platform (refer to section 2.4.3). Besides this, flat file importers are a common feature. Unfortunately a comprehensive approach abstracting from specific database management systems and database schemas is not provided by any of these yet.

In order to provide a generic object model for sequence data along with its related information (refer to section 2.1.2) and to allow unified access to the different existing sequence data sources, the *Unified Molecular Data Access (UMDA)* framework has been developed. In addition to the data access and management capabilities, *UMDA* offers a unified interface to access search index capabilities for primer/probe design and evaluation.

The following sections will introduce the concepts behind *UMDA* comprising the object model, the abstract interfaces for data selection and retrieval as well as for search index capabilities. Afterwards important implementation decisions are presented. This includes a short introduction to the incorporated flat file import and export capabilities as well as the integration of programmatic access to the *SRS* data integration platform (refer to section 2.4.2.3).

3.1.2 Object Model

The *UMDA object model* is divided into two parts: the *common objects* for ubiquitous purposes and the *specific objects* targeting mainly nucleic acid sequence data and its analysis.

3.1.2.1 Common Objects

Regarding the manifold possibilities to implement algorithms as well as libraries, a generic and versatile way of providing library or application specific settings is required. This helps to deal with for example several different libraries for connecting to database management systems (DBMS) providing a unified interface to all of them. The same holds true if different DBMS query systems are employed, for example the SQL language and a proprietary query system such as the one of the ARBDB (refer to section 2.4.1).

The *UMDA common objects* provide generic objects for query systems as well as application parameters and settings, summarized as traits. Figure 3.1 shows the two distinct object hierarchies, i.e. the query related objects and the trait related objects.

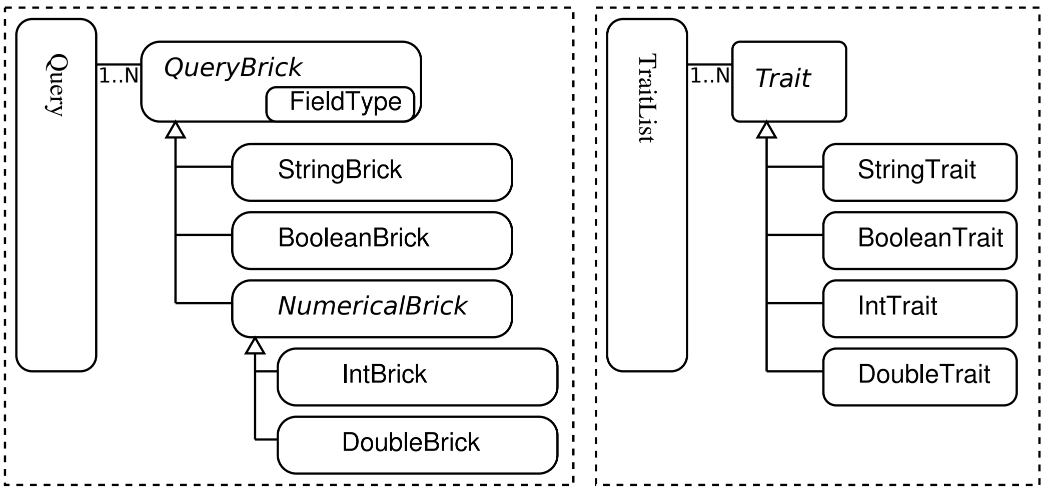


Figure 3.1: *UMDA Object Model* overview of the *common objects*

Query related objects facilitate the development of a generic query system based on the combination of simple building blocks which are independent of a specific query language. An all-text query representation is avoided. In addition, it allows easy programmatic processing relying on build-in data types of the targeted programming language C++.

This is achieved by a *Query* object which comprises a ordered list of one or more *QueryBrick* objects. The abstract *QueryBrick* object has four specific subtypes: *StringBrick*, *IntBrick*, *DoubleBrick* and *BooleanBrick*. All subtype objects have some information in common, i.e. the *FieldType* information. It comprises in particular an identifier of the field to search in, a human readable name or description of the field, its data-type (`string`, `boolean`, `integer` or `double`) and the link-type (`AND` and `OR`). The link-type specifies in which way a building block is combined with the previous building blocks.

Furthermore each subtype contains exclusive information. The *StringBrick*, suited for exact string comparisons and regular expressions, stores a string value and the comparison mode, i.e. `equal` or `not equal`. The *BooleanBrick* stores a boolean value in addition to the common information. The two numerical subtypes *IntBrick* and *DoubleBrick* store a type specific numerical value along with an operand. The operand is provided by the intermediate abstract type *NumericalBrick* and is either `equal`, `not equal`, `lower`, `greater`, `lower-equal` or `greater-equal`.

Trait related objects are stored within a *TraitList* container incorporating an arbitrary number of abstract *Trait* objects without a specific order. *Trait* objects are accessed by their name which functions as an identifier. The abstract *Trait* object has four specific subtypes: *StringTrait*, *IntTrait*, *DoubleTrait* and *BooleanTrait*. Besides a name field, all subtypes have a description field in common to provide human readable information. Furthermore each subtype stores specific information in form of a value of the appropriate type, for example a string for *StringTrait* objects. Additionally each *Trait* object contains a default value.

3.1.2.2 Specific Objects

The *UMDA object model* concept for *specific objects* shown in figure 3.2 provides the capabilities for representing annotated molecular sequences as well as phylogenetic data, clustering and structural information and results derived from in-silico analysis. The objects are classified into three categories: data, definition and result.

Data Objects The *SequenceEntry* object is the base container for a sequence and its related meta and annotation data. It comprises the most common fields provided by most data suppliers as direct members, for example a human readable name, a description, the sequence alphabet and the accession identifier among others. Information not directly integrated can be added using different associated objects including *Sequence*, *Feature*, bibliographic *Reference* and *Organism*. A *SequenceEntry* can have only one *Sequence* and *Organism* associated to it, but it may contain several *Features* and bibliographic *References*.

A *Sequence* object contains only the bare sequence as text-string without any further descriptive data. The idea behind the separation of *Sequence* and *SequenceEntry* is to allow a sequence string to be loaded only if required. In the context of whole genomes this is of great interest as a sequence string may get very large in size.

The *Organism* object represents the information of the organismal taxonomic classification, for example the genus and species.

The *Feature* object forms a container for sequence features such as genes. It provides fields for the type and name as well as associated objects for the reference and location information. The *Location* object holds the start and end position of a feature on the sequence as well as the sense.

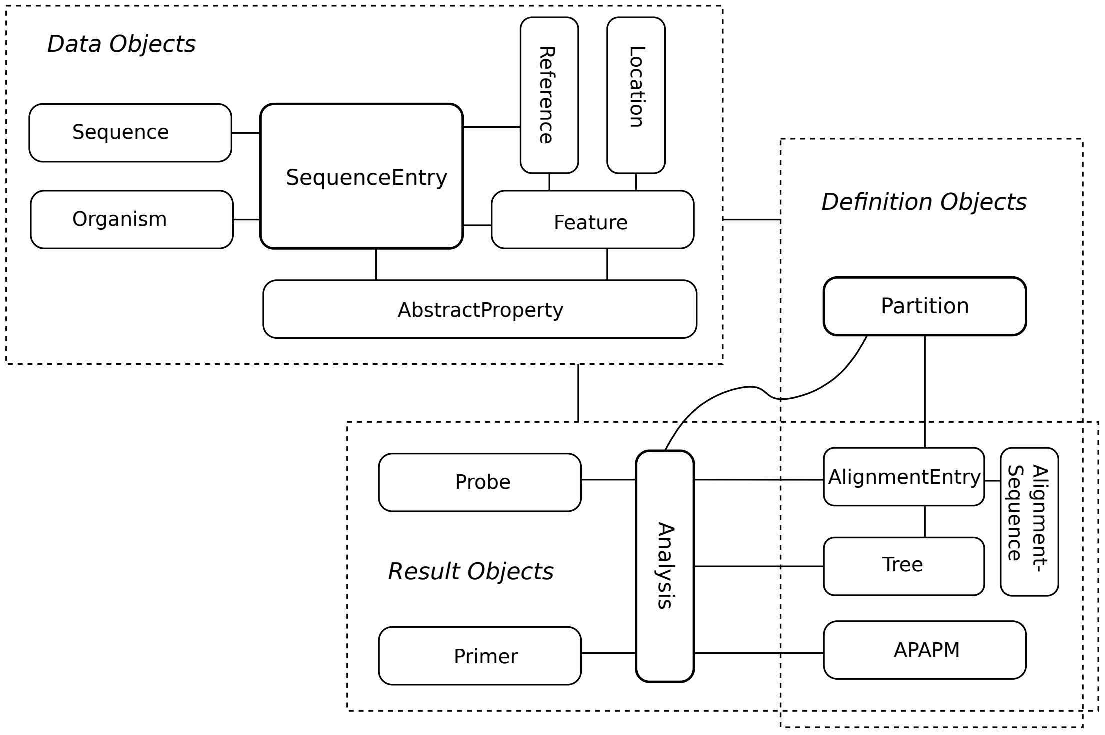


Figure 3.2: *UMDA Object Model* overview of the *specific objects*

Reference and *AbstractProperty* objects can be attached to either a *SequenceEntry* or *Feature* object. *Reference* objects include common literature information such as author, title and journal which is important to track the origin of data. *AbstractProperty* objects can be used to store arbitrary information in a key/value-like manner. An *AbstractProperty* can be utilized to store arbitrary additional information not covered by the subset of data fields available per default, for example to store internal and external database references or further comments.

In order to facilitate analysis based on selected sequence features, a combined identifier called *RichEntryId* has been added to the objects for convenient handling of *SequenceEntry-Feature* combinations. A *RichEntryId* can refer to a *SequenceEntry* only as well. In this case, the *Feature* identifier is left empty.

Definition and Result Objects *Partition* objects provide the facility to build arbitrary clusters of *SequenceEntry* objects, optionally selecting particular *Features*. The *SequenceEntry-Feature* combination is represented by a *RichEntryId* within each *Partition* object it is attached to. Furthermore it is possible to build groups inside a *Partition*. An entry of a *Partition* may belong to none or several groups. Special variants of the *Partition* object are *AlignmentEntry* and *Tree* objects. An *AlignmentEntry* object in conjunction with *AlignmentSequences* is capable of holding alignment information for each entry inside a partition in form of a 'Compact Idiosyncratic Gapped Alignment Report' (CIGAR) format string (refer to section 2.1.2). A *Tree* object is used for storing the partition in an acyclic graph. Usually it is a cluster representation such as a phylogenetic or other tree including group information at the inner nodes. The entries of the partition are located at the leaves of the tree. Molecular phylogenetic trees are usually based on a certain alignment. This relationship can be stored by referring from a *Tree* object to an *AlignmentEntry* object.

The *Alignment Position Associated Property Map (APAPM)* object is a container for a string which assigns an encoding to each position of an alignment along with the information which function to apply to the different string characters. The latter information is stored in form of a string which can be used to file the settings in an arbitrary text format. The information storable in an *APAPM* ranges from structural information, sequence region dependent biological, chemical or physical properties, accessibility patterns among other position dependent information. It can be used by applications in order to analyze specifically labeled alignment positions only, for example for visualizing special sequence segments such as primer binding sites. Another example application is a coloring of certain positions when displaying sequences of an alignment using the color scheme provided in the *APAPM* settings field.

For nucleotide acid sequence signatures there are the *Primer* and *Probe* objects. A *Primer* can hold a pair of signature sequence strings as well as information on the strand while a *Probe* can hold a single signature string.

The *Analysis* object is a container for linking information retrieved through computations with the data the analysis is based on, i.e. a *Partition*. The computed data ranges from *Primer* and *Probe* objects to defined or computed values such as *APAPM*, *AlignmentEntry* and *Tree* objects. All mentioned information can be used as input for computations as well. For example does this approach allow the storing of primers and probes in context of phylogenetic data. Furthermore the *Analysis* object provides a text field to store arbitrary information in a user defined format. This is useful to store for example the settings of external programs utilized for computations along with pre- and post-calculation steps.

3.1.3 Abstract Interfaces

In computer science, a widely used concept is to define one or more layers of abstraction to separate a concept from the specific instances. In addition, these abstraction layers offer levels of indirection utilized to reduce the complexity of solutions relying on it. A prominent example is the ISO/OSI network layer model where the programmer can rely on the highest layers for data transfer without the need to take care of for example the specific physical media participating.

Within *UMDA*, several unified interfaces have been defined in order to provide access to certain data and functionality without presenting implementation details. The aim is to provide a versatile and modular system which can be extended avoiding the need to adopt every algorithm and application build upon it if a new implementation of an interface is added.

The abstraction layers within *UMDA* are denoted as *abstract interfaces*. Abstract interfaces have been defined for data access and management, lazy-loading capabilities to defer costly data transfers as well as search index construction and utilization for primer/probe design and evaluation. These interfaces will be described in the next sections.

3.1.3.1 Lazy Object Loading Interface

The *lazy-loader interface* targets the deferring of load operations of data. This is of interest for the objects in the UMDA object model which are organized in a hierarchical structure, for example the *SequenceEntry* with its child objects including *Sequence* and *Feature* objects.

To avoid unnecessary load operations if some child objects are not required in conjunction with providing a convenient way to retrieve this data later, the abstract *LazyLoadObject* has been defined. It provides general purpose method interfaces for loading child objects as well as methods to check if they have been retrieved already.

The *LazyLoadObject* is derived by the *SequenceEntryLazyLoader* and the *AlignmentEntry-LazyLoader* abstract interfaces taking care of the specific requirements of *SequenceEntry* respectively *Alignment* objects.

3.1.3.2 DataBase Interface (DBI)

The *UMDA database interface (DBI)* defines a generic interface for querying and accessing molecular sequence and annotation data within a database. It consists of three interrelated functional groups of methods building sub-application programming interfaces: the *ManagementAPI*, *AccessAPI* and *QueryAPI* which are independet of the utilized database management system (figure 3.3).

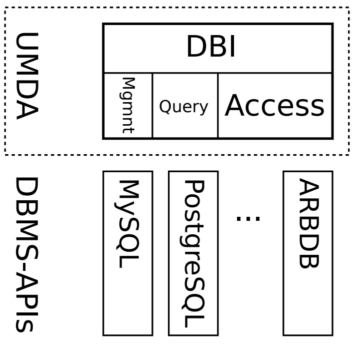


Figure 3.3: *UMDA DBI* schema comprising the *ManagementAPI*, *AccessAPI* and *QueryAPI*

The *ManagementAPI* provides generic database management methods for setting up the connection properties, connecting and disconnecting to the database. For limiting accessibility to certain databases, there is a read-only mode with the appropriate setup and check functionality.

As not all settings of a specific program or library can be easily abstracted in a generic way, for example some DBMS access libraries may support encrypted transfer while others do not, custom traits for non-generic settings are utilized. A *TraitList* comprising default *Trait* objects (refer to section 3.1) can be obtained. The altered values can be returned to the *DBI* which takes care of updating the settings. An implementation may return an empty *TraitList* if there are no custom settings required or possible.

The modular *AccessAPI* offers methods for loading, storing and deleting of single data objects and so called *bulk*-methods working on lists for handling several objects at once.

For objects with children, for example *SequenceEntry* along with its associated *Sequence*, *Organism*, *Features*, *References* and *AbstractProperties*, it is possible to either cascade the operations or perform them on the single object only.

The interface is centered around database identifiers as keys which are provided by the methods of the *QueryAPI*. Given a valid database identifier as parameter, the load method returns and the delete methods erases a single object. In contrast, the store methods require a reference to the object to store as input. The *bulk*-methods work on appropriate lists of either database identifiers or object references instead of the single parameters.

Due to its modular design, it is not required to implement the *AccessAPI* for every object of the object model. In order to facilitate the programmatic checking for supported objects, appropriate method interfaces are declared.

The *QueryAPI* is designed to be independent of any specific type of database or query language. It imposes no restrictions concerning the database management system types. It provides methods for retrieving lists of database identifiers, either without any constrains or by a search based on a *Query* object with its incorporated *QueryBricks* (refer to section 3.1). Furthermore the *QueryAPI* offers methods for retrieving the available *FieldType* information required for the *QueryBricks* of a query.

3.1.3.3 Search Index Interface (SII)

The *UMDA search index interface (SII)* defines a generic interface for search index based oligonucleotide pattern matching as well as primer/probe design and evaluation capabilities. It is divided into two sub-APIs: the *ManagementAPI* and the *ApplicationAPI* (figure 3.4).

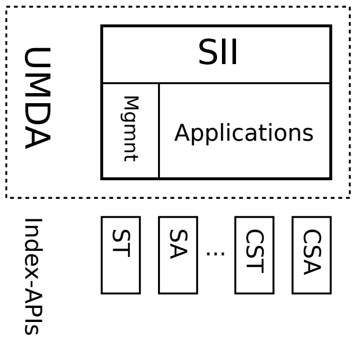


Figure 3.4: *UMDA SII* schema comprising the *ManagementAPI* and *ApplicationAPI*

The *ManagementAPI* provides index construction and load capabilities. Construction can be conducted based on one or more selected *SequenceEntry* objects which are retrieved from an *UMDA DBI* or a *multiFASTA* file. The index is stored under a provided name in a user specified directory.

In addition, optional functions for adding *SequenceEntry* objects to or removing them from the index are available. Furthermore methods to access sequence data entries incorporated into the index are defined. Based on the original data source identifier of a sequence entry, its incorporation within the index can be checked as well.

For program or library specific settings, custom traits for non-generic settings are utilized. A *TraitList* comprising default *Trait* objects (refer to section 3.1) can be obtained. The altered values can be returned to the SII which takes care of updating the settings. An implementation may return an empty *TraitList* if there are no custom settings.

The *ApplicationAPI* provides functions for the main targeted field of application, i.e. oligonucleotide pattern matching, primer/probe design and evaluation capabilities and sequence similarity query functionality.

For *oligonucleotide string matching* an interface taking query settings as input (table 3.1) and returning a match-list (table 3.2) has been defined. Further functions are available to check for the existence or the number of occurrences of a pattern only. The query settings comprise various parameters besides the pattern to match. One parameter is the maximum allowed distance of a match, i.e. the number of errors allowed. In addition it is possible to individually allow only certain types of errors, namely substitutions, insertions and deletions. Furthermore a flag allows to switch between weighted and non-weighted approximate string matching. The number of "N"-mismatches can be restricted and it is possible to query the index for the reverse pattern optionally. Finally there are parameters to influence the returned hit information, i.e. to optionally include a differential alignment or sequence feature information if available.

The result of a query is returned as a list offering detailed hit information. For each hit this is the distance, position and the number of wildcards, substitutions, deletions and insertions. In addition the position relative to a reference entry and optionally hit information related to sequence features as well as a differential alignment are returned.

description	type
pattern	string
max distance	double
reverse search	boolean
weighted search	boolean
allow substitution	boolean
allow insertion	boolean
allow deletion	boolean
set max wildcards	integer
create diff alignment	boolean
feature mode	boolean

Table 3.1: *UMDA SII* string matching settings

description	type
identifier	*RichEntryId*
position	integer
reference position [opt]	integer
distance	integer
weighted distance	double
number of wildcards	integer
number of substitutions	integer
number of insertions	integer
number of deletions	integer
reverse hit	boolean
differential alignment [opt]	string
general hit info [opt]	string
feature hits [opt]	map

Table 3.2: *UMDA SII* string matching result values

Furthermore the *ApplicationAPI* defines interfaces for index based primer/probe design functionality. One method allows to retrieve all oligonucleotides up to a given length. In addition there is an interface to access primer/probe design capabilities based on several parameters (table 3.3). For a user defined target group, passed as a list of identifiers, the desired primer/probe candidate length, group coverage and maximum number of out-group hits can be defined. In addition parameters for providing a melting temperature range and borders for the GC-content are provided.

The results of the design process are returned as a list (table 3.3), which can be optionally truncated. For each designed primer/probe candidate the list comprises the number of in-group hits as well as the out-group hits for an increasing number of allowed weighted mismatches. In addition a quality score in conjunction with the melting temperature estimation and the GC-content are included.

description	type
target identifiers	map
length	integer
min group coverage (%)	double
max non-group hits	integer
min GC-content (%)	double
max GC-content (%)	double
min melting temperature	double
max melting temperature	double
min position	integer
max position	integer
max number of candidates	integer
sort mode	integer

Table 3.3: *UMDA SII* design settings

description	type
primer/probe candidate	string
length	integer
in-group hits	integer
melting temperature	double
GC-content	integer
relative position	integer
reference position	integer
quality	double
out-group hits [wmis]	vector

Table 3.4: *UMDA SII* design result values

description	type
sequence	string
window size	integer
fast mode	boolean
max distance	double
search mode	integer
range start	integer
range end	integer
max return number	integer
sort mode	integer

Table 3.5: *UMDA SII* similarity settings

description	type
identifier	*RichEntryId*
number of hits	integer
hit percentage	double

Table 3.6: *UMDA SII* similarity result values

Finally, the *ApplicationAPI* defines an interface for index supported determination of sequence similarity based on several parameters (table 3.5). A similarity search can be conducted for a given sequence which is divided into sub-patterns of a length given as window size. Additional parameters are the maximum allowed distance for each pattern as well as options to influence the algorithm, i.e. switches to influence the type of search to conduct. This can be utilized to reduce the number of queries by omitting patterns or to conduct several queries for a single pattern by reversing, complementing or reverse-complementing it. It is also possible to narrow down the sequence region of interest to a specific target range in order to ignore all query hits lying outside the range.

The results are returned in a list which can be optionally sorted or truncated. The values returned by a similarity search are defined as a combination of the *RichEntryId* and two score values, i.e. the absolute and the relative number of hits (table 3.6).

3.1.4 Implementation

The implementation of the *UMDA* framework design was influenced by some of the following paradigms for good application programming interfaces (APIs):

- make it easy to use (by a programmer)
- make it consistent
- avoid 3rd-party dependencies if possible
- make it modular

C++ was chosen as programming language as it is said to be among the fastest ones, although the programmer must be careful to write code which is readable, maintainable and contains no errors. Other reasons include the possibility to utilize C code from within C++, for example MPI libraries (refer to section 2.6.3.3). This makes it possible to utilize high performance computing environments besides common desktop systems.

In order to make the utilization of the *UMDA* framework for a programmer easier, the implementation relies heavily on the *resource acquisition is initialization (RAII)* idiom thought up by Bjarne Stroustrup, the C++ inventor. With *RAII* the resource management, i.e. to a major part the memory allocated, is coupled with a runtime object. *RAII* is used in conjunction with reference counted pointers, also known as smart pointers. Smart pointers free the object they refer to automatically if the counter reaches 0, relieving the programmer from many manual tasks when writing a program.

Another simplification for programmers is provided in form of a serialization interface for all appropriate object types of the *UMDA object model* as well as the objects defined for the *abstract interfaces*. For example objects can be serialized for storing the information in a file and loading it back into a runtime object. This also enables transferring objects via a network and deserializing them on the receiving side with minimal effort.

The *UMDA* framework consists of a mandatory core comprising the *UMDA object model* and the *abstract interfaces*. The *UMDA* core implementation relies on the standard C++ STL functionality and the peer-reviewed Boost libraries [18] only.

Specific implementations of the abstract interfaces as well as applications may depend on further 3rd-party libraries and can be switched on or off individually prior to the *UMDA* framework build. The utilized optional libraries comprise the *mysql++* library [98], the *pqxx* library [80] as well as the *ARBDB API* (refer to section 2.4.4.2) and the *hwloc* library [59]. Finally for GUI components, the *Qt* framework [115] has been utilized.

3.1.4.1 Object Model

The whole *UMDA object model* implementation makes use of smart pointers to enable cascading deallocation of objects if a root object, i.e. for example a *SequenceEntry*, is not referenced any more. Furthermore, this allows programmers to work on a single copy of an object in memory instead of copying and synchronizing it all the time. All objects offer a `clone()` method for obtaining a copy which does not rely on its origin any more. Furthermore, all objects implement a serialization interface based on the `boost::serialization` library.

The *UMDA specific objects* (refer to section 3.1.2.2) are derived from a base object with generic functionality. It comprises a callback system as well as object status flags. The callback system allows to broadcast information up and down the object hierarchy. The basic object offers status flags for indicating if an object is valid, modified, removed or marked.

3.1.4.2 DBI And SII Plugins

The *UMDA* interfaces for accessing databases (*DBI*) and search index functionality (*SII*) are kept abstract to avoid the dependency on specific libraries. In order to further facilitate the independent and generic design approach, the implementations are encapsulated into plugins. Each plugin is based on the facade design pattern [40] presenting only the *UMDA DBI* for accessing the underlying DBMS, respectively the *UMDA SII* for accessing a search index, to the programmer. In particular this is utilized to hide functionality of the libraries employed for the implementation which is not required, for example the direct access to the SQL query methods of a RDBMS library.

The plugin mechanism allows to utilize the abstract interfaces without requiring to recompile the whole project if a new plugin is added. Hence the *DBI* and *SII* implementations are decoupled from the core project making the dependency unidirectional, i.e. the plugins rely on the core project but not the other way round.

DBI The *UMDA database interface (DBI)* has been implemented for various database schemas and database management systems. Parallel to the *DBI* development, a new *UMDA-MySQL* genome database schema has been developed providing capabilities to store the *UMDA specific objects* (section 3.1.2.2) in a MySQL database. Besides this new *UMDA-MySQL* schema, none of the other exisiting genome sequence data schemas (refer to section 2.1.3) supports the full range of *UMDA* objects. Fortunately the modular design of the *UMDA DBI* allowed the implementation of at least the supported objects which included for all schemas the *SequenceEntry* hierarchy. An overview over the available *DBI* plugins is given in table 3.7.

The *UMDA IMDB* (In-Memory DataBase) is a special *DBI* implementation based on the standard C++ STL container objects. It can be utilized for temporal in-memory storage and for testing purposes. The *DBI* implementation for the Chado schema in conjunction

DB schema	DBMS	DBMS API	object support
ARB	ARBDB	ARBDB API	partly
BioSQL	MySQL	mysql++	partly
Chado	PostgreSQL	pqxx	[prototype]
EnsEMBL	MySQL	mysql++	partly
UMDA-MySQL	MySQL	mysql++	full
UMDA IMDB	n/a	C++ STL	full

Table 3.7: *UMDA DBI* plugins overview

with the PostgreSQL DBMS is currently only available as prototype showing the principal functionality.

SII The *UMDA search index interface (SII)* has been implemented prototypically with help of the Seqan C++ template library [30, 44]. As it is meant as a proof-of-concept, this *SeqanSII* prototype implements only the *SII* string matching interface. For approximate string matching (ASM) Seqan offers online algorithms, but no implementation based on the available index structures (personal communication with Manuel Holtgrewe on Seqan mailinglist).

A complete implementation of the *SII* has been conducted for the *PTPan library* (refer to sections 3.2 and 3.3). Custom traits are utilized for non-generic construction and application settings, for example the verbose mode, the number of threads to use and the optional inclusion of sequence features into the index.

3.1.4.3 DataSet

The *DataSet* is a proof-of-concept implementation of an in-memory management system to handle the complete access to a database over a *DBI*. It is a control instance taking care of loading, storing and deleting objects. Object modifications are only stored to the corresponding database on request. The methods exposed to the programmer allow the retrieval and removal of root objects like *SequenceEntry* or *Tree* only. The object callback system is used to pass over the information about the removal of a child object to the *DataSet* which keeps track of all modifications, i.e. new and removed objects. To make them persistent or revoke them, methods to save or clear are available.

3.1.4.4 Flat File Importers And Exporters

Several flat file importers and exporters have been implemented in the *UMDA* framework to provide an easy way to load data from the common formats into the *UMDA object model*. For *multiFASTA* files one or more *SequenceEntry* objects along with the corresponding *Sequence* are returned while for *EMBL-Bank* flat files the full related hierarchy is populated with the loaded data, for example the *Feature* objects.

The importers and exporters are designed to take a C++ stream as input source. This allows to utilize them for parsing input data from different sources, for example files or network streams.

3.1.4.5 SRS Integration

In order to allow access to the *SRS* integration platform (refer to section 2.4.2.3) from within C++ applications, a library has been developed by E. Cai under my supervision [20].

The library handles the communication with the *SRS* server comprising the retrieval of available databases and the field lists for each database. During this process, the obtained HTML-files are transformed into an XML representation omitting all unnecessary information. With this information it is possible to build queries which can be submitted to the *SRS* server utilizing the wgetz REST API. The results can be retrieved in different formats like HTML or in different flat file formats such as *FASTA* or the *EMBL-Bank* flat file format (refer to section 2.1.4). If the data is retrieved in an appropriate flat file format, it can be directly forwarded to a *UMDA* flat file parser (refer to section 3.1.4.4). This allows the direct import of data into *UMDA* based applications without the need to store flat files to a filesystem first.

This library has been incorporated into the *UMDA* framework. Based on the libraries capabilities, a GUI employing the Qt-framework has been developed. It allows to query the *SRS* system and retrieve the sequence data.

3.1.5 Optimization

The *UMDA* framework has been optimized to facilitate the utilization in parallel environments as well as to ease the management of data in main memory.

For parallel environments, internal counters are based on atomic integers to avoid the application of explicit locks like mutexes (refer to section 2.6.3.2).

Further improvements comprise the reuse of database connections by utilizing a connection-pool as well as a unique object identifier and a callback system to allow passing information from child objects to its parent. This simplifies the management of objects in main memory and is utilized for example by the *DataSet* presented above (refer to section 3.1.4.3).

In addition each object can be transformed into a serialized representation which allows to store and reload them in text files or to transfer them via network as stated before.

Many other improvements were integrated during design and implementation as well. The whole framework was developed in a cyclic agile process.

3.1.6 Summary

The *Unified Molecular Data Access (UMDA)* framework provides a generic object model along with abstract interfaces for database access and search index based primer/probe design and evaluation capabilities.

The *database interface (DBI)* enables direct access and querying of molecular sequence data located in different primary databases. This allows to retrieve data in order to build and curate customized local secondary databases or for further processing by applications. The conversion of the different data representations is done implicitly by the interface.

Furthermore *UMDA* allows to store sequence and sequence group related information such as phylogenetic and secondary structure data as well as primer and probe sequences and properties derived from computations in one integrative database. In addition it offers the possibility to arbitrarily create partitions on the data within the database.

The *search index interface (SII)* enables utilization of search index based primer/probe design and evaluation capabilities. The index can be constructed based on selected sequence entry data retrieved from a *DBI* or *multiFASTA* file. Oligonucleotide pattern matching, primer/probe design and similarity search can be conducted based on user defined parameters returning meaningful result lists.

An overview of the implemented *UMDA* framework with its layers is presented in figure 3.5.

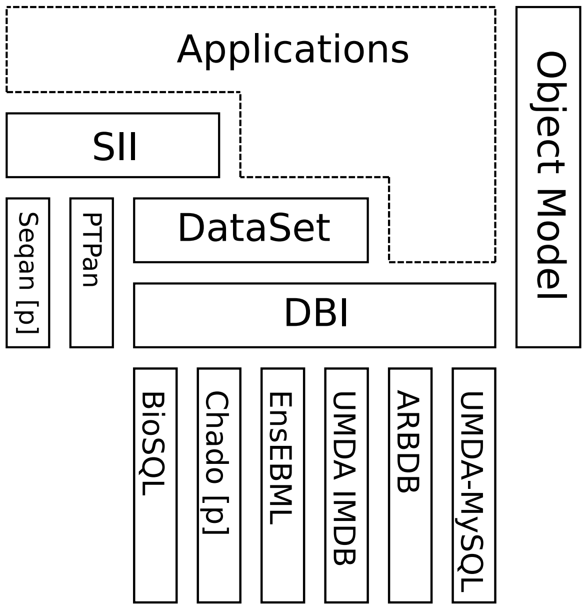


Figure 3.5: *UMDA* complete framework overview

Several prototypical and ready-to-use programs as well as the *SRS* library integration and the flat file parser have been developed. These prototypes include command line applications as well as graphical interfaces to demonstrate the functionality provided by the *DBI* and *SII*.

In addition, a fully functional graphical tool, the *UMDA Primer/Probe (UPP) Designer*, has been developed. It allows to either construct an index with a *DBI* or *multiFASTA* file as data source or to load a pre-build index. The index is accessed utilizing the *SII* and all applications for primer/probe design and evaluation provided by the *ApplicationAPI* are supported by appropriate graphical interfaces. More details are provided later in section 3.5.

Finally to facilitate the development and easy regression tests, several unit tests have been implemented for the whole *UMDA object model* as well as the abstract *DBI*. For each implemented *DBI* only the unit tests for the supported object types are carried out.

3.2 PTPan Index Structure

In the following sections first the motivation to develop *PTPan* is presented followed by the index structure design. Finally the construction algorithm is described.

3.2.1 Motivation

As revealed by the statistics provided by the Genome OnLine Database (GOLD) [81], the amount of available genome data, mostly for bacteria, is increasing at a rapid pace as shown in figure 1.1. The same holds true for gene databases like the collections of small subunit ribosomal RNA (ssu-rRNA) such as Greengenes [29] or SILVA [113]. The fast growth will go on in the next few years as the latest sequencing technologies, which produce large amounts of molecular sequence data in short time, are becoming standard [85, 90].

This genome sequence data is of great interest concerning in-silico analysis and computational molecular diagnostics (CMD). CMD is the in-silico search for molecular markers, primers and probes based on sequence and phylogenetic data and can lead to a faster development of molecular detection methods for pathogens while reducing the experimental cost in the wet lab.

Requirements Primer/probe design applications based on nucleic acid gene or genome sequence data often utilize indexing structures to speed up computations [73, 84, 111]. Several requirements are imposed on the index structures in order to conduct proper up to date primer/probe design and evaluation. In order to keep up with the increasing amount of sequence data, the index should be operable even when the main memory size is limited.

Furthermore, as the sequence data may contain possible sequencing errors or mutations, it is necessary to identify insertions or deletions (indels) to avoid misjudgment of the sensitivity of primers or probes. This would otherwise have severe implications to experimental results as stated by McIlroy and colleagues for fluorescence in situ hybridization (FISH) [89]. Thus the index should be able to efficiently carry out approximate string matching based on the Levenshtein-distance metric (refer to section 2.2.3.2).

The handling of ambiguous sequence characters and treating matches at ambiguous positions in an efficient way is of interest as well. This can be achieved by treating all ambiguities as wild-card in string matching (refer to section 2.2.2).

Furthermore approximate string matching should support a weight scheme (refer to section 2.2.4). Weighting approximate matches with respect to type and position of mismatches can help to scale the distance between exact and inexact matches as well as to better rate the specificity and sensitivity of a probe as stated by Yilmaz and colleagues [140].

For primer/probe evaluation, the incorporation of alignment data should be supported as well. Beyond the precise comparative positioning of matches, it provides the opportunity to include higher order structure and function information, such as probe accessibility, in the in-silico design process as discussed by Kumar and colleagues [72].

Existing solutions A wide range of different index based solutions exist to design primer and probes based on genome sequences (refer to section 2.4.4.3).

The *CMD/PSID* is capable of identifying signatures based on sequence collections. Unfortunately it can conduct the design process only for single sequences, but not for whole groups of sequences. This is a major drawback limiting its field of application. Designing primers and probes for detecting and distinguishing organism groups rather than single organisms is required in applications such as microbial population analysis and molecular screening for microbial pathogens or indicators.

Two approaches based on pre-processed collections are the web-based *Insignia* server and *CaSSiS*. Both allow to select candidates based on clustering information the pre-processing is based on. Designing own candidates on-the-fly is not possible without repeating the compute intensive pre-processing of the whole collection with different parameters set.

With the ARB PT-Server, incorporated in the ARB software environment [84], a suffix tree based search index providing many of the aforementioned requirements for oligonucleotide primer/probe design exists. It is capable of conducting approximate string matching in weighted- and non-weighted mode. In addition it respects the wildcard character "N" and takes into account alignment information if available.

The major drawbacks of the PT-Server are that it has to fit into main memory during construction and application. Although memory requirements are reduced by utilizing truncation of the suffix tree, the PT-Server is not capable of dealing with the rapidly growing databases due to still high main memory demands. In addition, the PT-Server relies on Hamming-distance metric for approximate string matching concerning primer/probe design and evaluation. Thus it is not capable of identifying indels which would require the Levenshtein-distance metric. In addition, the PT-Server can be constructed from an ARB database only. Furthermore some index based functionality requires the source database being available during application. This limits the field of application of the PT-Server strictly to the ARB environment.

The problem of increasing amounts of genome sequence data has been tackled by many state-of-the-art indexing techniques. These techniques, mainly originating from indexing structure theory, are of interest as the core structure for new developments.

Several in-memory solutions exist. To reduce its memory requirements, the k-truncated suffix tree (*kTST*) [120] limits the height of the suffix tree like the PT-Server does. Hence it suffers from the same memory constraints. The enhanced suffix array (*eSA*) invented to replace suffix trees, has reduced memory requirements, but still needs to fit into main memory entirely during application [1]. The so called self-indexes combine a compressed representation of the source text with an index structure. By this they are capable of reducing the memory requirements to the size of the original input text [102]. In the field of short read mapping, which is related to approximate oligonucleotide string matching, self-indexes have been successfully employed in tools like *BWA* [78], a representative mapping tool known to be among the fastest ones. Unfortunately, according to Russo and colleagues, approximate string matching based on self-indexes faces severe slowdowns compared to classical indexes like suffix trees or arrays [116].

Besides the aforementioned in-memory solutions, several approaches try to deal with the large amount of data by utilizing secondary storage during index construction and application. The disk resident suffix arrays (*rSA*) offers moderate disk space requirements and it provides exact string matching [92]. Unfortunately it is not capable of conducting approximate string matching. Another well studied approach are suffix trees in external memory (*eST*), recently reviewed by Barsky and colleagues [8]. *eST* can be constructed with limited main memory efficiently, although memory requirements on secondary storage are high. Furthermore, an open challenge for the existing *eST* approaches is the utilization for approximate string matching.

In addition several approaches exist to leverage the partitioning of *eST* to construct standard suffix trees on secondary storage utilizing parallel shared as well as distributed memory systems (refer to section 2.6.4).

C. Hodges investigated in his diploma thesis the general possibility of employing compression techniques for *eST* to handle collections of gene sequences like the SILVA database [55]. He developed a functional prototype with reduced memory requirements capable of performing Levenshtein-distance metric based approximate string matching. Unfortunately it is not capable of dealing with genome data. Based on C. Hodges results, J. Böhnel analyzed in his diploma thesis the possibilities of parallel index construction in distributed memory environments resulting in a proprietary prototypic client-server application [17]. Furthermore he investigated the applicability of his solution for oligonucleotide primer/probe design.

Evaluation result Based on this evaluation of existing approaches, with Levenshtein-distance metric based approximate oligonucleotide string matching as main purpose in mind, *eST* seems the most appealing approach. Nevertheless it has remaining challenges, namely the memory consumption on secondary storage and the inability to perform approximate string matching reasonably. The high memory requirements may be partly addressed by combining *eST* with truncation employed in *kTST* and the PT-Server. In addition a compression scheme can be employed. In addition, the partitioning employed by *eST*s can be leveraged to speed up construction on parallel architectures.

3.2.2 Structure Design

After evaluating the requirements for a nucleic acid sequence data index and reviewing the existing solutions, *PTPan* was designed to fulfill the following major goals:

- data source independence
- low main memory demands
- effective Levenshtein-distance based *ASM*
- persistent storage

Targeting nucleic acid genome sequence data with its lack of delimiters, the index is based on a partitioned and truncated suffix tree on secondary storage. It is stored in a compressed format which is optimized to allow DFS based algorithms to perform reasonably fast even though the index is larger than main memory and stored on common hard disk drives. This enables effective Levenshtein-distance based ASM as well as complete DFS-order tree traversal required for oligonucleotide primer/probe design applications to perform well. To obtain independence from the sequence data source, all required data is incorporated into the index. A previous version of *PTPan* targeting the efficient handling of curated gene sequence databases like the one provided by the SILVA project (refer to section 2.1.5) has been published in [32].

The index is separated into individual parts: the index header and one or more partitions each corresponding to a prefix. The suffix tree for each specific prefix is stored compressed within a partition. The next sections describe the structures and the compression in detail.

3.2.2.1 Index Header

The index header contains the basic information about the index. It comprises meta information like the truncation depth, the number of sequence entries included as well as the total number of nucleic acid bases incorporated. If available, a reference sequence entry is incorporated including its sequence string.

To obtain independence from the sequence data source, for each entry the nucleic acid sequence is stored in compact representation (refer to section 2.6.1). If the sequence contains alignment information, this is incorporated as well. Besides the sequence string, the original database identifier as well as a description are stored for each entry. To allow faster decompression of random sequence parts, *jump labels* are incorporated. These allow to start decompressing a sequence at a specific position and omit sequence parts not required. Additionally, genome sequence feature information can be included into the index header optionally, i.e. an identifier referring to its source database entry and the range on the genome within it is located. Finally the index header includes the full list of index partitions like filenames and prefix information.

3.2.2.2 Suffix Tree Stream Compression

The suffix tree stream-compression serves two purposes: reducing the memory requirements and allowing efficient approximate string matching on secondary storage.

The compression is obtained by storing the in-memory suffix tree nodes as compressed nodes. The order of the nodes is in depth-first-search (DFS) order starting with the root node. Two kinds of nodes are distinguished: inner and border. Inner nodes have references to other inner or border nodes, i.e. outgoing edges, while border nodes contain the suffix occurrences as leaf-array instead.

Each compressed node consists of three consecutive parts:

- ingoing edge label
- branch mask (refer to section 2.6.2)
- outgoing references (inner nodes) or leaf-array (border nodes)

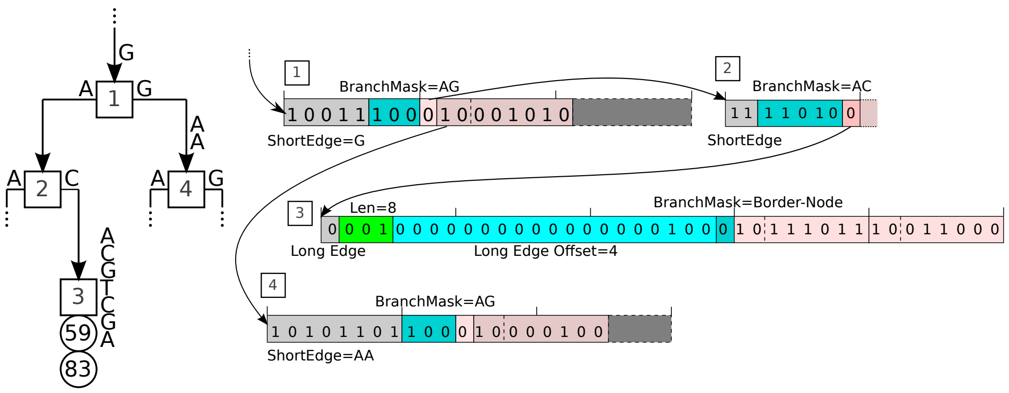
The only exception is the root node which has no in-going edge. For all other nodes, the compressed in-going edge label differs depending on the length of the edge. Edges up to a certain threshold are denoted as *short edges* if their label occurs often enough in the whole suffix tree. Scarce short edges as well as all edges exceeding the threshold value are denoted *long edge*. In order to reduce the average size of the corresponding representations, Huffman encoding (refer to section 2.6.2) is utilized for the *short edges*, the lengths of the *long edges* as well as the *branch masks*.

In the stream-compression, nodes with a short in-going edge are initiated by their appropriate short edge Huffman code, i.e. the edge label is stored directly. The length of the in-going edge is given implicitly by the Huffman encoding which comprises the original label and its length in the mapping table. In contrast the compressed representation for long edges is headed by the code identifying long edges followed by the Huffman encoded edge length. Instead of storing the edge label directly, a dictionary string is utilized. Hence in the suffix tree stream the next value is the dictionary offset. The long edge label sequence is the concatenation of the first base defined by the branch mask of the parent node (refer to section 2.6.2) and the remaining *length-1* bases from the dictionary string.

Following the edge information, for inner nodes the branch mask Huffman code is issued. This is succeeded by up to five encoded child reference offsets in DFS order corresponding to the branch mask bits. As the offsets are written in ascending order, they are stored utilizing delta encoding (refer to section 2.6.2).

For border nodes the child references are replaced by the leaf-array. A special code for the branch mask is utilized to indicate border nodes. The suffix occurrences are sorted in ascending order and delta encoded to store the leaf-array more efficiently.

A detailed example of the suffix tree stream compression is shown in figure 3.6. It shows a suffix tree excerpt with its corresponding stream representation as well as the relevant data structure excerpts.



	Short Edge HuffCode	
Long Edge	0	0
Len=1	1	11
N	8	10101000111
A	9	10000
C	10	101101
G	11	10011
T	12	101110
AA	38	10101101
	...	...

	Branch Mask HuffCode	
BNode	1	0
NA	3	11100011
NC	5	11100001
AC	6	11010
NAC	7	11100000000
NG	9	1100001
AG	10	100
NAG	11	110000001
	...	...

Long Edge Length HuffCode	
3	1010111011
4	101011100
5	1010110
6	1011
7	000
8	001
9	100
10	111
...	...

Long Edge Dictionary
ACGTACGTCGACA...

Offset width 19 bits

Child References	
Dist=0	0
Dist=x	10.x^6
Dist=x	110.x^11
Dist=x	111.x^27

Figure 3.6: *PTPan* suffix tree excerpt with corresponding stream-compression (derived from [32] supplementary figure 1) [numbering of the tree nodes for clarification only]

Decompressing a stream-compressed suffix tree can be done by inverting the compression steps. First the ingoing edge information is read and decompressed followed by the branch mask. Afterwards for inner nodes the child references and for border nodes the leaf-array are decompressed.

3.2.2.3 Index Partitions

Each index partition has the same structure. It contains meta information about the index part it represents as well as the corresponding suffix tree with all data required for decompressing. The meta information comprises the prefix of the partition, an identifier to check if it corresponds to the partition information stored in the index header as well as the partition index size. The stream-compressed suffix tree is stored along with the Huffman encodings for short edges, long edge length and branch masks. Furthermore the long edge dictionary is stored including the maximum number of bits required by a long edge offset.

3.2.3 Construction Algorithm

The *PTPan* index construction algorithm is divided into three main steps:

1. data retrieval (refer to section 3.2.3.1)
2. data preparation (refer to section 3.2.3.2)
3. index construction (refer to section 3.2.3.3)

The next sections will give insights on all three major steps.

3.2.3.1 Data Retrieval

The first step of the index construction is the data retrieval and incorporation into the index header (algorithm 1).

Algorithm 1 *PTPan* data retrieval pseudocode

```
[...]
header = initializeIndexHeader();
while data_interface.hasNextEntry() do
    ptpanEntry = data_interface.getNextEntry();
    ptpanEntry.compactSequenceData();
    header.appendAndUpdateGlobalState(ptpanEntry);
end while
header.storeGlobalState();
[...]
```

The distinct sequence data entries and optional a reference entry as well as the related feature data, are retrieved over a defined abstract interface. It can be implemented for different sources and is currently available for the *ARBDB API*, *multiFASTA* files and the *UMDA DBI*.

After an entry has been retrieved, the sequence is transformed into its compact representation (refer to section 2.6.1). The jump labels are determined during this process, too. Optionally available alignment information, i.e. gap characters, are incorporated as well.

All other non-*DNA5* characters are treated as "N" character. Afterwards, the complete entry information is stored directly into the index header on disk.

Each time an entry is added to the header, the global counters for the number of entries and the total number of sequence bases are updated. For each entry, the start position in a concatenation of all sequences is stored, i.e. the sum of all *DNA5* bases retrieved so far. After all data has been retrieved, these global counters are stored into the index header.

3.2.3.2 Data Preparation

Before the index can be constructed, the gathered data needs to be pre-processed.

Merge sequences The first step is to merge the individual sequences of all retrieved entries into a single temporary raw data file in compact representation (algorithm 2). All gap characters, if available, are stripped during this process. In order to ease the construction algorithm (refer to section 3.2.3.3), the raw merged sequence is finally padded with "N"-characters at the end.

Algorithm 2 *PTPan* merge sequences pseudocode

```
[...]
temporaryMergedData = initMergedData();
while header.hasNextEntry() do
    ptpanEntry = header.getNextEntry();
    temporaryMergedData.stripGapsAndAppend(ptpanEntry);
end whiletemporaryMergedData.padWithN();
[...]
```

Partition determination The second preparation step is to determine the index partitions depending on the amount of main memory available and the total number of bases in the merged raw data (algorithm 3).

The partitions are defined by variable length prefixes. Prior to prefix calculation, the maximum number of base positions fitting into the available main memory is determined by a *worst case estimation*. If the size of the temporary merged raw data is lower than the estimated value, only one partition is required.

Otherwise the partition prefixes must be calculated. First the data-dependent maximum prefix length is determined. Second a scan over the merged raw data is conducted counting the occurrences of all possible prefixes of this exact length. The prefixes are stored in a histogram with their compact representation as key, i.e. an integer value (refer to section 2.6.1). This allows to determine the number of occurrences of each prefix of a prefix by adding the occurrences within a numerical key range. Taking the fact that $5^n > \sum_{i=0}^{n-1} 4 * 5^i$ the range start and end can be determined as follows: Range start is the character followed by only "N"s, so the numerical value is $val(\text{char}) * 5^{n-1}$ with n being the prefix length. The compact representation utilizes $n-1$ down to 0 as powers for the base. Range end is the next characters range start value minus one. Some of the numerical values in between

Algorithm 3 *PTPan* partition determination pseudocode

```
[...]
maxPartitionSize = determineMaxPartitionSize(memorySize,tempMergedDataSize);
if maxPartitionSize < tempMergedDataSize then
   finalPartitions = setupOnePartition();
else
   maxPrefixLength = determineMaxPrefixLength(memorySize,tempMergedDataSize);
   histogram = scan(temporaryMergedData, maxPrefixLength);
   initialPartitions = init(prefixLength = 1);
   while NOT initialPartitions.empty() do
      part = initialPartitions.popFirst();
      if histogram.size(part) $\leq$ maxPartitionSize then
         finalPartitions.add(part);
      else
         initialPartitions.pushBack(part.refine());
      end if
   end while
end if
header.storePartitionInformation(finalPartitions);
[...]
```

do not represent valid sequences in compact representation. This is no problem as the number of occurrences for them will be zero. For the last range, the end is given by the fact that the highest value is lower than 5^n in any case. To illustrate this by an example assume a prefix of all "T"s, which results in the highest possible numerical value for a compact representation. For a prefix length of five, the compact numerical representation is $\sum_{i=0}^{5-1} val(\text{“T”}) * 5^i = 3124$. With $5^5 = 3125 > 3124$ the upper border can be determined easily.

After building the histogram, a partition candidate of prefix length one is initialized for each symbol of the alphabet, i.e. five for the *DNA5* alphabet, and added to a queue. Next, while the queue is not empty, the first candidate is removed. With help of the histogram, the current candidate is checked if it fits into main memory during construction, i.e. the number of bases in the partition does not exceed the threshold value. If the check succeeds, the current partition candidate is put into the final partition list. Prefixes not occurring in the merged raw data are omitted. Otherwise, if the check fails, the partition is refined. It is extended by one base for each symbol of the alphabet, generating new partition candidates, again five for the *DNA5* alphabet. The newly defined partition candidates are added to the end of the queue. Then the iterative process continues with the next partition in the queue until no candidates are left.

Figure 3.7 shows an example partition refinement for a maximum prefix length of five. The question marks denote any of the five *DNA5* bases. The ranges are determined as described above. The refinement is done for "G" by applying the range determination for prefix length 4 and taking the already determined range start as base counter. For the example nine partitions are in the final list.

Finally, the partition information is stored into the index header before the partition

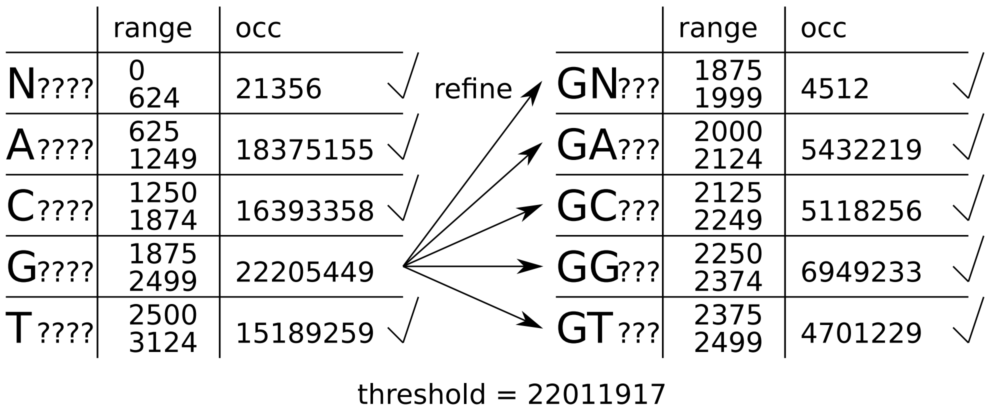


Figure 3.7: *PTPan* example partition prefix determination with histogram ranges and number of occurrences [maximum prefix length is 5]

construction starts.

3.2.3.3 Index Construction

After all data has been prepared and the final index header has been stored, the index partitions are constructed and stored to disk (algorithm 4).

Algorithm 4 *PTPan* construction pseudocode

```
[...]
while header.hasNextPartition() do
  partition = header.getNextPartition();
  tree = initTree();
  currentWindow = mergedRawData.begin();
  while currentWindow != mergedRawData.end() do
    if currentWindow.prefix() == partition.prefix() then
      tree.insertOccurrence(currentWindow);
    end if
    currentWindow.shift();
  end while
  tree.prepareEdgesAndBranchMasks();
  tree.buildLongEdgeDictionary();
  tree.relocateReferences();
  tree.writeToDisk();
end while
```

The first step for each partition is the construction of a truncated suffix tree in main memory for the given pruning length. The in-memory tree structures correspond to the structures of the stream-compressed tree representation introduced in section 3.2.2.2. Inner nodes and border nodes are distinguished with their specific layout, i.e. having child references (inner nodes) or a leaf-array (border nodes). For in-going edges the compact representation as well as its length are kept.

Insert prefix occurrence In order to build up the in-memory suffix tree for a partition, the merged raw data sequence is scanned once. To avoid repeated access to the temporary file, a sliding window keeps a sufficiently long part of the raw data in main memory to obtain the window sequence string of length equal to the pruning depth starting at the current position. This sequence is checked if its prefix matches the partition prefix. If so, the in-memory suffix tree is traversed down comparing the edge labels with the window sequence.

If a border node is reached during this process, the current occurrence is added as a new leaf because the path denoted by the window sequence is already present in the suffix tree. If a mismatch appears inside an inner node, a new border node including the current occurrence as leaf is added. The remaining sequence window part not matching the path traversed down so far is the in-going edge label of this new border node. In contrast, if the mismatch lies inside an edge label, splitting the latter is required. A new inner node is added to the suffix tree with the common sequence part as in-going edge and the downstream node of the edge split is updated. Finally, a new border node with one leaf is added. Its in-going edge is the remaining part of the current window sequence.

The algorithm continues by shifting the current window sequence by one base either if it does not match the partitions prefix or after inserting is finished.

In order to speed up the construction algorithm, a hash-map is utilized to store the suffix-prefixes up to a certain length. The compact numerical representation functions as key. The payload is a pointer to the appropriate inner node which is reached by traversing the in-memory suffix tree from the root down the path defined by the characters of the prefix. If a new window sequence is added to the suffix tree, the algorithm first checks if its prefix is contained in the hash-map. If it is already present, the comparison can continue at the corresponding inner node avoiding several edge comparisons.

Prepare edges and branch masks After building the in-memory suffix tree, it must be prepared before transforming it into the stream-compressed representation (refer to section 3.2.2.2). First the short edges as well as the frequency of branch mask combinations are determined by traversing the in-memory suffix tree and counting the occurrences. Afterwards the Huffman encodings for short edges and branch masks are calculated. The short edge encodings contain a single value to identify long edges as well. A second traversal is conducted to count the long edges and to replace the short edges by their Huffman representation.

Building the long edge dictionary After all long edges are known, the construction algorithm utilizes a greedy algorithm to build the long edge dictionary.

The first step is to determine all unique long edges by adding each long edge to a hash-map with its compact representation as key. The first time a long edge key is added, the reference to the suffix tree node is stored as payload. If for any subsequent long edge label the compact representation is already present in the hash-map, the suffix tree node is marked to contain a non-unique edge. The edge label is then updated to point to the first occurrence respectively the corresponding unique node. This reduces the total number of edges to process as the same edge label may occur several times in the suffix tree.

As next step, all unique long edge labels given by the hash-map's key-value pairs are sorted in lexicographical order. Equal edges and those which are prefix to other ones are grouped together and their merged raw data offsets are equated.

A second sort arranges the unique edges by their merged raw data offsets. With the prior equation step, it is now possible to detect overlapping fragments. These can be concatenated into longer chains for the dictionary covering many of the smaller edge fragments.

Afterwards all concatenated long edges are added one after another to the dictionary string. Next the references of all long edges are updated to point to the dictionary string offset instead of the merged raw data offset. This is done in two iterations, the first for the unique long edges and the second for the long edges pointing to a unique one.

Finally, the long edge dictionary string is transformed into its compact representation and a Huffman encoding is built for the long edge lengths.

Relocating the tree The last step prior to writing the suffix tree into its stream-compressed representation is the relocation of the suffix tree references. In the final representation, the downward child pointers of an inner node are stored as delta encoded offsets in the stream relative to the position right after the current node. This requires that the size of all child nodes must be available before storing a node. Hence the tree must be traversed in depth-first-search (DFS) order once before writing in order to determine the size of each node, i.e. the sub-tree size for which it is the root. During this traversal, each downward child pointer of an inner node is replaced by the size of the corresponding child node sub-tree. As the in-memory tree structure is lost by this procedure, the DFS order is kept separately in an array of pointers.

Writing the stream-compressed suffix tree The suffix tree is written to disk in its stream-compressed representation (refer to section 3.2.2.2). Each node in the DFS-order array generated in the relocation step is written one after another into a stream of bits. This stream is stored into a file.

3.3 PTPan Applications

Based on the *PTPan* index structure, several applications supporting nucleic acid genome sequence based molecular diagnostics have been developed.

The next sections will describe the basic *PTPan* tree traversal algorithm. It is the base for the different implemented applications for primer/probe design and evaluation as well as sequence similarity searches which are described afterwards.

3.3.1 Basic PTPan Tree Traversal

The basic *PTPan* suffix tree traversal is the starting point of the developed applications. For each partition, the suffix tree in its stream-compressed representation is recursively traversed top-down in depth-first search order (algorithm 5).

Algorithm 5 *PTPan* application: basic tree traversal pseudocode

```
[...]
root = decompressRoot();
traverse_rec(root);
[...]
———traverse_rec(node)———
if node.isInnerNode() then
  while node.hasNextChild() do
    child = node.decompressNextChild();
    traverse_rec(child);
  end while
else
  handleBorderNode(node);
end if
———traverse_rec()———
```

First the root is decompressed as it is located at the beginning of each partition suffix-tree-stream (refer to section 3.2.2.2). Like all inner nodes, the decompressed root node reveals the branch mask as well as the available offsets to the child nodes. With this information, each child node can be decompressed and processed. The order is thereby given by the branch mask also indicating which subtrees are missing compared to a fully populated tree. If a border node is reached, the recursion stops. An algorithm relying on the basic traversal can now handle the border node according to its requirements.

3.3.2 ProbeMatch

ProbeMatch is a oligonucleotide string matching algorithm based on the basic tree traversal (refer to section 3.3.1). It is capable of conducting exact as well as approximate string matching (ASM) (algorithm 6).

Algorithm 6 *PTPan ProbeMatch* pseudocode

```
[...]
for all partitions do
  if partition.prefixRelevant() then
    root = partition.decompressRoot();
    search_rec(root);
  end if
end for
results.verifyHits();
results.createDifferentialAlignment();
results.sort();
[...]
———search_rec(node)———
node.checkRemainingPattern();
if NOT node.reachedMaxErrors() then
  if node.reachedPatternEnd() then
    results.gatherSubtreeOccurrences(node);
  else
    if node.isInnerNode() then
      while node.hasNextChild() do
        child = node.decompressNextChild();
        search_rec(child);
      end while
    else
      results.verifyAndAddOccurrences(node);
    end if
  end if
end if
———search_rec()———
```

Depending on the settings, ASM can be based either on the Levenshtein- or the Hamming-distance metric (refer to section 2.2.3). Furthermore, ASM is available in either a *basic-match-mode* with uniform error values or in a *weighted-match-mode* (refer to section 2.2.4). In both modes, the proper treatment of "N"s as source sequence wildcards is supported (refer to section 2.2.2).

The index partitions are searched one after another. The results are merged into a single list. For each node, the pattern and the ingoing edge label are compared. For the Hamming-distance metric this is done character by character while for the Levenshtein-distance metric a comparison matrix is build. For inner nodes the branch mask is utilized to check the first character of the downward edge before traversing the tree further down. If the maximum error rate is not exceeded and the pattern end is not reached, the recursion continues for inner nodes with the child nodes while it stops for border nodes.

If the pattern matches the path down to a node or the pruning depth is reached without exceeding the error limit, the occurrences are added to the match list. For inner node, this requires the gathering of all occurrences by traversing further down to the border nodes. Each hit crossing an entry border in the merged raw sequence data is filtered out before adding it to the list. If the pattern exceeds the pruning depth, it is marked in order to verify it later by comparing it to the decompressed original sequence from the index header. This enables searching for patterns which exceed the pruning depth of the suffix tree.

The final steps prior to returning the match list are to optionally sort the list and to build a differential alignment (refer to section 2.1.2) including in addition the sequence context of the match at nine positions at 3′ and 5′ ends.

The match list returned comprises individual values for each hit (table 3.8). These values are the entry database identifier, a short human read-able information and the number of mismatches, i.e. the individual counter for substitutions, insertions and deletions. Furthermore the weighted mismatch value, the number of "N"-mismatches as well as the hit position within the sequence and optionally in relation to a reference entry are returned. Optionally the differential alignment with the sequence context is included for each hit as well.

entry-id	entry identifier
entry-info	entry information
mis-sub	substitution count
mis-ins	insertion count
mis-del	deletion count
wmis	weighted mismatch value
nmis	N-mismatches
pos	position
refpos [opt]	reference entry position
diff-align [opt]	differential alignment & context

Table 3.8: *PTPan ProbeMatch* return values

3.3.3 ProbeDesign

ProbeDesign is a primer/probe design algorithm implemented based on the *PTPan* basic tree traversal and ProbeMatch functionality (algorithm 7). The aim is to determine unique oligonucleotide signature sequences as well as the resulting primer/probe candidate (refer to section 2.1.2), i.e. to find signatures with a high coverage for a selected single sequence or group of sequences.

If available in the index, individual genome sequence features can be selected as target group for primer/probe design instead of the whole genome sequence as well. The corresponding subsequence stretch defined by the feature range will be treated as the selected sequence.

Algorithm 7 *PTPan ProbeDesign* pseudocode

```
[...]
for all partitions do
    root = partition.decompressRoot();
    results.gather_rec(root, length);
    candidateList.append(results);
end for
for all candidateList do
    probeMatch(candidate);
end for
candidateList.calculateQuality();
candidateList.sort();
[...]
———gather_rec(node, length)———
if node.reachedDepth(length) then
    results.gatherVerifyAddSubtreeOccurrences(node);
else
    if node.isInnerNode() then
        while node.hasNextChild() do
            child = node.decompressNextChild();
            gather_rec(child, length);
        end while
    end if
end if
———gather_rec()——–
```

Preferably the selected group should be hit by signatures entirely while avoiding out-group hits or at least hitting non-group sequence entries only with a high distance. Further parameters allow to constrain the search. These are the melting temperature, GC-content and probe length (refer to section 2.4.4.1).

First the algorithm gathers all signature candidates of the targeted length which must not exceed the pruning depth of the suffix tree. For each partition the suffix tree is traversed based on the basic tree traversal. If the path of a node has a sufficient length, a candidate has been found. Up to this step, the functionality is available as standalone function as well. This allows to obtain signatures of a given length contained in at least one sequence entry within the index.

Before adding a path to the candidate list, it is checked for the temperature and GC-content constrains. If not outruled, the occurrences of the path are gathered to check the number of in- and out-group hits. Only if passing all checks, the candidate is appended to the final candidate list.

After gathering all candidates a *ProbeMatch* is conducted in *weighted-match-mode* for each candidate. The predefined maximum mismatch value is 4.0. From the match result list, an array of out-group hit numbers for increasing weighted mismatch values in steps of 0.2 is obtained. This array is the base for the calculation of the signature sequence quality value.

Finally the candidate list is returned with individual values for each primer/probe candidate (table 3.9). Besides the candidate signature stretch and the corresponding primer/probe candidate, the GC-content and a melting temperature estimation are returned. In addition, the quality value and the array of out-group hit numbers is included each individual candidate.

candidate	primer/probe candidate
signature	corresponding signature
G+C	GC-content
temperature	melting temperature estimation
quality	quality value
out-grp-wmis	out-group hit numbers for increasing wmis values

Table 3.9: *PTPan ProbeDesign* return values

3.3.4 SimilaritySearch

SimilaritySearch provides a method for sequence similarity determination by oligonucleotide string matching frequencies (algorithm 8). The aim is to find the most similar sequences in a dataset without the requirement to conduct more complicated analysis, for example calculating an alignment.

Algorithm 8 *PTPan SimilaritySearch* pseudocode

```
[...]
list.init();
sequenceWindow.init(sequence);
while NOT sequenceWindow.endOfSequenceReached() do
  pattern = sequenceWindow.shift();
  result = probeMatch(pattern);
  list.update(result);
end while
[...]
```

For an input sequence, all oligonucleotide substrings of a predefined length are generated by shifting a window frame. For each substring obtained this way, *ProbeMatch* based exact or approximate string matching is performed, depending on the presets. The number of hits are summed up individually for each sequence entry available in the *PTPan* index. From the total number and the number of *ProbeMatch*es performed, a hit percentage value is calculated. Finally the two result value types are returned as a list of individual *SimilaritySearch* values for each sequence entry in *PTPan* (table 3.10). According to the settings, the list is truncated and sorted by either decreasing hit score or hit percentage.

entry-id	entry identifier
numHits	number of hits
hitPercent	hit percentage

Table 3.10: *PTPan SimilaritySearch* return values

3.4 PTPan Optimization And Parallelization

In his widely known article "The Free Lunch Is Over" Herb Sutter stated in 2005 that it is necessary to leverage parallelism of modern processor architectures to speed up the runtime of a program [132]. The automatic performance gain due to increasing clock rates is not given any more since the advent of multi-core processors.

In this thesis, for all optimization and parallelization efforts, the initial step has been a manual code review. This was followed by profiling the code to obtain the parts consuming the majority of the overall runtime. This has been done based on test gene and genome sequence data sets of different sizes, i.e. the overall number of bases. The results have been utilized to improve the sequential code as well as for parallelization efforts to remove bottlenecks.

3.4.1 Construction Algorithm

The results of different profiling runs were utilized to improve the sequential construction code and to parallelize the construction algorithm on shared and distributed memory systems. The main results are presented in the following sections.

3.4.1.1 Sequential Optimization

Several improvements have been made compared to the previously published version of *PTPan* [32]. They are already incorporated in the previous chapters and not mentioned explicitly again, for example the improved long edge dictionary build process (refer to section 3.2.3.3).

During profiling of the construction process for different sequence data sets, the main memory requirements showed to be significantly lower than the worst case memory requirement estimation (refer to section 3.2.3.2 - partition determination). This is the case if many of the sequence data entries are equal for a significant amount of bases, i.e. they are homologue. This is the case for highly repetitive databases like the 16S rRNA gene database SILVA (refer to section 2.1.5).
To adopt the construction algorithm to the actually reduced main memory requirements, the first addition was an optional output of statistical information about the memory requirements during construction, most important the memory usage rate in percent compared to the worst case estimation. Second an optional *memory ratio* construction parameter has been added. It allows to define the memory utilization ratio with respect to the worst case prior to index construction. The memory ratio is respected during partition determination by multiplying the worst-case estimation with it. This results potentially in a lower partition count and with this in a reduced construction time.

3.4.1.2 Shared Memory Parallelization

The shared memory parallelization efforts are based on `boost::threads` in combination with the `boost::threadpool` for thread lifetime management. *OpenMP* was neglected to ease portability and to avoid requirement of an *OpenMP* aware compiler. In contrast, the `boost` libraries can be compiled together with *PTPan* if necessary as they depend only on common available libraries. Other options like *Intel Threading Building Blocks (TBB)* have been omitted to avoid adding a new dependency. *PTPan* already utilized different data structure related `boost` libraries before parallelizing the construction algorithm.

Profiling revealed the construction of the partitions being the most time consuming part of the construction algorithm followed by the data retrieval and preparation. Thus the following sections present the conducted efforts in these parts of the complete construction algorithm. The modifications compared to the sequential algorithms are highlighted by printing them italic in the algorithm pseudocode.

Data retrieval To enable parallel data retrieval, the implementation of the abstract interface must be thread-safe. A single sequence entry can be loaded utilizing a thread-safe retrieve method. Afterwards it is prepared by transforming the sequence into the compact representation. Finally the index header is locked for writing the entry and updating the global state variables (algorithm 9).

Algorithm 9 *PTPan parallel* data retrieval pseudocode

```
[...]
header = initializeIndexHeader();
for all threads do
  while interface.threadSafe_hasNextEntry() do
    ptpanEntry = interface.threadSafe_getNextEntry());
    ptpanEntry.compactSequenceData();
    header.lock();
    header.appendAndUpdateGlobalState(ptpanEntry);
    header.unlock();
  end while
end for
header.storeGlobalState();
[...]
```

Data preparation The first step is to merge the individual sequences of all retrieved entries into a single temporary raw data file in compact representation (algorithm 2). For every set of two consecutive sequence entries, the end of the first and the beginning of the second sequence share an overlapping region in the final temporary file. Writing these overlapping parts must be synchronized while there is no need to do this for the non-overlapping parts, i.e. the majority of the data to write (algorithm 10).

For each sequence entry in the header to add to the merged raw data, the start position in the merged raw data and the length of the sequence are known beforehand. Hence it is possible to calculate the number of overlapping bases at the start and the end of each sequence.

To synchronize the writing of an overlapping part, a map is utilized. The entry number of the sequence ending in an overlapping region is taken as key while the compact representation of the overlapping part contributed by one of the adjacent sequences is the value. The first thread reaching an overlapping end puts its part into the map after locking it. The second thread can now retrieve the part, combine it with its part of the overlapping region and write it to the merged raw data.

Algorithm 10 *PTPan parallel* merge sequences pseudocode

```
[...]
temporaryMergedData = initMergedData();
for all threads do
   while header.threadSafe_hasNextEntry() do
      ptpanEntry = header.threadSafe_getNextEntry();
      overlapStart = ptpanEntry.getStart();
      if overlapStart then
         checkAndAdd(overlapStart);
      end if
      temporaryMergedData.appendNonOverlapping(ptpanEntry);
      overlapEnd = ptpanEntry.getEnd();
      if overlapEnd then
         checkAndAdd(overlapEnd);
      end if
   end while
end for
[...]
———checkAndAdd(overlap)———
overlapMap.lock();
if overlapMap.hasCorrespondingPart(overlap) then
   combined = overlapMap.getCorrespondingPartAndCombineWith(overlap);
   temporaryMergedData.insertOverlapping(combined);
else
   overlapMap.insertPart(overlap);
end if
overlapMap.unlock();
———checkAndAdd()——-
```

Partition determination In order to support the parallel construction of partitions on a shared memory system, the partition determination algorithm has been adopted. The largest partitions to construct must fit into memory in parallel.

First the partitions are determined in the same way as for a single thread, based on the same worst case memory requirements estimation. Afterwards the new algorithm checks if the largest partitions up to the number of threads utilized fit into main memory in parallel. If not, the largest partition is removed from the list and refined, i.e. the prefix is extended by one character for each symbol of the alphabet. Afterwards the memory requirements for the new partitions are determined. This process is repeated until the largest partitions fit into memory in parallel.

Algorithm 11 *PTPan* partition determination *for parallel* pseudocode

```
[...]
maxPartitionSize = determineMaxPartitionSize(memorySize,tempMergedDataSize);
if maxPartitionSize < tempMergedDataSize then
  finalPartitions = setupOnePartition();
else
  maxPrefixLength = determineMaxPrefixLength(memorySize,tempMergedDataSize,
  threadCount);
  histogram = scan(temporaryMergedData, maxPrefixLength);
  initialPartitions = init(length = 1);
  while NOT initialPartitions.empty() do
    part = initialPartitions.popFirst();
    if histogram.size(part) $\leq$ maxPartitionSize then
      finalPartitions.sortIn(part);
    else
      initialPartitions.pushBack(part.refine());
    end if
  end while
  while finalPartitions.sumLargest(maxThreadCount) > maxPartitionSize do
    finalPartitions.refineLargest(histogram);
  end while
end if
header.storePartitionInformation(finalPartitions);
[...]
```

Optionally, the algorithm for the thread-count based refinement can be altered by a parameter. Instead of always taking the maximum number of threads, the partition distribution and thread-count providing the best *partitions-per-thread ratio* are taken. In order to achieve this, the best combination is kept separately during refinement (algorithm 12). After evaluating all thread-counts, construction is performed based on this best combination.

Partition construction The partition construction itself is an embarrassing parallel problem (refer to section 2.6.3.1). The partitions are independent and the global structures like the merged raw data are accessed reading only. The partitions fit into main memory in parallel which is ensured by the partition determination algorithm for parallel construction. Thus each participation thread constructs one partition after another as long as there are candidates remaining in the list. Only the retrieval of the next partition candidate in the list to construct requires synchronization.

3.4.1.3 Distributed Memory Parallelization

In order to utilize cluster computers to speed up *PTPan* index construction, a distributed memory construction algorithm has been developed and implemented based on the *Message Passing Interface (MPI)* (refer to section 2.6.3.3).

Algorithm 12 *PTPan* partition determination *for parallel* optional pseudocode

```
[...]
tmpPartitions = remember(finalPartitions, 1);
for threadCount := 2 to maxThreadCount do
    while finalPartitions.sumLargest(threadCount) > maxPartitionSize do
        finalPartitions.refineLargest(histogram);
    end while
    if tmpPartitions.ratio() > finalPartitions.ratio() then
        tmpPartitions = remember(finalPartitions, threadCount);
    end if
end for
finalPartitions = tmpPartitions.partitions();
threadCount = tmpPartitions.threadCount();
[...]
```

Algorithm 13 *PTPan parallel* construction pseudocode

```
[...]
for all threads do
    while header.threadSafe_hasNextPartition() do
        partition = header.threadSafe_getNextPartition();
        tree = initTree();
        currentWindow = mergedRawData.begin();
        while currentWindow != mergedRawData.end() do
            if currentWindow.prefix() = partition.prefix() then
                tree.insertOccurrence(currentWindow);
            end if
            currentWindow.shift();
        end while
        tree.prepareEdgesAndBranchMasks();
        tree.buildLongEdgeDictionary();
        tree.relocateReferences();
        tree.writeToDisk();
    end while
end for
```

The participating computing nodes are split up into one master node and all other nodes as slaves. The master is responsible for handling the data retrieval and preparation as well as the setup of the slaves. In particular the master distributes the merged raw data and the information required to construct a suffix tree partition. In contrast, the slaves are responsible for the construction of the individual partitions. After a slave finished constructing a partition, it sends statistical information back to the master and receives the next partition to construct. If no more partitions are to be done, the master sends a done message to all slaves and gathers the remaining statistics. The slaves shut down after receiving the done message. Finally the master cleans up remaining structures, for example the merged raw data file.

Detailed pseudocode of the master and the slave is presented in algorithm 14 and 15.

Algorithm 14 MPI *PTPan* construction: master pseudocode

```
[...]
retrieveAndPrepareData();
MPI_Barrier_1.wait();
MPI_Bcast(send, generalIndexData);
MPI_Barrier_2.wait();
for all slaves do
  partitionInfo = nextPartition();
  MPI_Send(partitionInfo);
end for
while partitionsLeft() do
  MPI_Receive(slaveConstructionResult, slaveId);
  partitionInfo = nextPartition();
  MPI_Send(slaveId, MORE_WORK);
  MPI_Send(slaveId, partitionInfo);
end while
for all slaves do
  MPI_Receive(slaveConstructionResult);
  MPI_Send(DONE);
end for
MPI_Barrier_3.wait();
cleanUp();
[...]
```

The implementation prototype developed reads the input data from a *multiFASTA* file and assumes a shared file system for all cluster nodes. Furthermore it assumes that each processing unit has an equal amount of memory available.

3.4.2 Applications

The different applications for primer/probe design and evaluation relying on the *PTPan* index structure have been profiled prior to optimization as well.

Algorithm 15 MPI *PTPan* construction: slaves pseudocode

```
[...]
MPI_Barrier_1.wait();
MPI_Bcast(receive, generalIndexData);
MPI_Barrier_2.wait();
while more-work do
  MPI_Receive(flag);
  if flag != DONE then
    MPI_Receive(partitionInfo);
    sequentialPartitionConstruction(partitionInfo);
    MPI_Send(constructionResult);
  else
    more-work = false;
  end if
end while
MPI_Barrier_3.wait();
[...]
```

ProbeMatch The differential alignment generation showed to be a time consuming part when carrying out *ProbeMatch* searches. One reason was the accessing and decompressing of the original sequence data. It took a long time for genome sequences as they required to be decompress from the beginning for each hit. By incorporating *jump labels* (refer to section 3.2.2.1) large parts of the compact sequence could be omitted during decoding as this process now starts closer to the region of interest near the hit position.

Second, as the different query hits are independent from each other, the creation of the differential alignment is an embarrassing parallel problem. It can be parallelized by distributing the processing of the hits to different threads of execution. The distribution is not done block-wise to avoid random loading of original sequence data from the index header. Each thread processes each $ith + threadCount$ hit with i being the thread number. As the hits are sorted in ascending order, the sequence entries are accessed in the same order they are stored in the index header preventing random access patterns.

Algorithm 16 *PTPan ProbeDesign parallel* prototype pseudocode

```
[...]
for all threads do
  while candidateList.threadSafe_hasMore() do
    candidate = candidateList.threadSafe_getNext();
    probeMatch(candidate);
  end while
end for
[...]
```

ProbeDesign The *ProbeDesign* algorithm includes a source of embarrassing parallelism in the loop conducting the *ProbeMatches*. A prototypical implementation leverages this (algorithm 16, based on algorithm 7 in section 3.3.3). It is currently suited for *PTPan* indexes fitting into main memory only. Parallel on-disk index access results in unfavorable access patterns.

SimilaritySearch The *SimilaritySearch* algorithm includes a source of embarrassing parallelism in the loop conducting the *ProbeMatches*, too. A prototypical implementation has been implemented to leverage this (algorithm 17, based on algorithm 8 in section 3.3.4). The hit counter for each entry has been adopted to be thread-safe by utilizing an atomic counter (refer to section 2.6.3.2). This algorithm suffers from the same limitation as the parallel *ProbeDesign* algorithm, i.e. it is currently suited for *PTPan* indexes fitting into main memory only.

Algorithm 17 *PTPan SimilaritySearch parallel* prototype pseudocode

```
[...]
list.init();
for all threads do
  offset = threadCount;
  sequenceWindow.init(sequence, offset);
  while NOT sequenceWindow.endOfSequenceReached() do
    pattern = sequenceWindow.shift(threadCount);
    result = probeMatch(pattern);
    list.threadSafe_update(result);
  end while
end for
[...]
```

3.5 Software Components

Besides the *UMDA* framework implementation (refer to section 3.1.4), several software components have been developed based on the algorithms described in the last chapters. In the next sections, the *PTPan* library and its integration into the ARB software environment and the *UMDA* framework are presented. Afterwards the integrated system for primer/probe design and evaluation based on the *UMDA* framework with its plugins for database access and search index application capabilities is described.

3.5.1 PTPan Library

Based on the developed *PTPan* structures, construction algorithm and applications, a standalone C++ library has been developed and implemented for 64 bit Linux systems.

The library incorporates an implementation of the abstract PTPan data retrieval interface for *multiFASTA* files which is provided as default data source for index construction. The memory size is detected automatically or can be manually passed as parameter. Another parameter is the depth of the suffix tree which defaults to 20. The maximum depth is currently limited to 27 due to the implementation decisions to rely on a single integer value for the edge label of an in-memory suffix tree node. In compact representation the maximum number of bases which can be stored in a 64-bit integer value are the aforementioned 27 bases for the DNA5 alphabet (refer to section 2.6.1). The library supports shared memory multi-threading to speed up construction and application of the index. The number of system hardware threads is determined automatically, although the thread-count to use can be varied by a parameter.

The three *PTPan* applications (refer to section 3.3) can be accessed utilizing appropriate methods. The application results are returned in index independent data structures for further utilization.

Furthermore, the library offers methods to obtain the database identifier list of the incorporated sequence entries. The entries can be accessed as well in order to gain access to the sequence in compact representation or the optionally available sequence features. This is of interest for *ProbeMatch* conducted on a genome index incorporating feature information. The feature hits are not returned directly. Instead, the feature hits can be obtained in a second step after the match list has been returned. The *PTPan* sequence entry corresponding to a hit provides a function to obtain all features for a given position and length.

3.5.2 ARB PTPan Integration

The *PTPan* C++ library with its primer/probe design and evaluation functionality has been integrated into the ARB software environment to complement or replace the ARB PT-Server (refer to section 2.4.4.2). In order to access the nucleic acid sequence data in an ARB database, the abstract *PTPan* data retrieval interface has been implemented for the ARBDB API. In addition a wrapper has been implemented for the functions of the PT-Server adopting the *PTPan* interface to the corresponding PT-Server calls. This allows to use *PTPan* as a drop-in replacement. Hence it is not required to modify any PT-Server based application of the ARB software environment in order to leverage *PTPan* functionality.

3.5.3 UMDA PTPan Integration

Based on the *PTPan* C++ library the *UMDA* search index interface has been implemented and integrated into the *PTPan-SII* plugin.

As data source for nucleic acid sequence and annotation data either a *multiFASTA* file or a *UMDA DBI* can be utilized. For *multiFASTA* files the *PTPan* library relies on the integrated data retriever implementation. A *PTPan* data retrieval interface for the *UMDA DBI* has been added.

The abstract *SII ApplicationAPI* has been implemented by mapping the different methods to the corresponding *PTPan* library application functions. This incorporates the conversion of the settings and the result lists between the *PTPan* library and the *UMDA* framework format.

3.5.4 UMDA Primer/Probe Designer

The *UMDA Primer/Probe (UPP) Designer* is a graphical user interface (GUI) for primer and probe design and evaluation based on the *UMDA* framework. It has been developed to provide easy to use search index construction from heterogeneous sources as well as *UMDA SII* based primer/probe design application capabilities. An workflow overview is given in figure 3.8.

For a user utilizing the *UPP Designer* the first step is to choose whether to construct and load a new index or to load an existing one. An index can be constructed from a *multiFASTA* file or a selectable *UMDA DBI* plugin as input source. In both cases the *UMDA SII* to utilize can be selected among the available *SII* plugins. If a *DBI* has been chosen as input source, it is possible to either construct an index for all entries available or to utilize the *QueryAPI* of the *DBI* to choose a subset of entries of the underlying database. Besides constructing a new index, it is also possible to load an existing index which can be chosen by a selection dialog.

After either loading or constructing an index, the GUI provides access to the three applications provided by the *UMDA SII ApplicationAPI* (refer to section 3.1.3.3).

The pattern matching functionality is provided by a single window which provides access to the settings as well as the results at once.

The graphical primer/probe design application interface is split into three parts. First a selection dialog allows to choose the target group of entries for primer/probe design. The target group can be selected manually or optionally it can be loaded from a file containing the database identifiers one per line as well. In the following second dialog window, the settings like signature length, maximum allowed out-group hits and coverage can be adjusted. In the last dialog window, after conducting the design process, the result list is shown. In order to ease further evaluation, it is possible to launch the pattern matching functionality window for a selected primer/probe candidate which is set as query pattern automatically.

Finally, the *UPP Designer* provides access to the similarity search functionality by a dialog window for the settings and the result list at once.

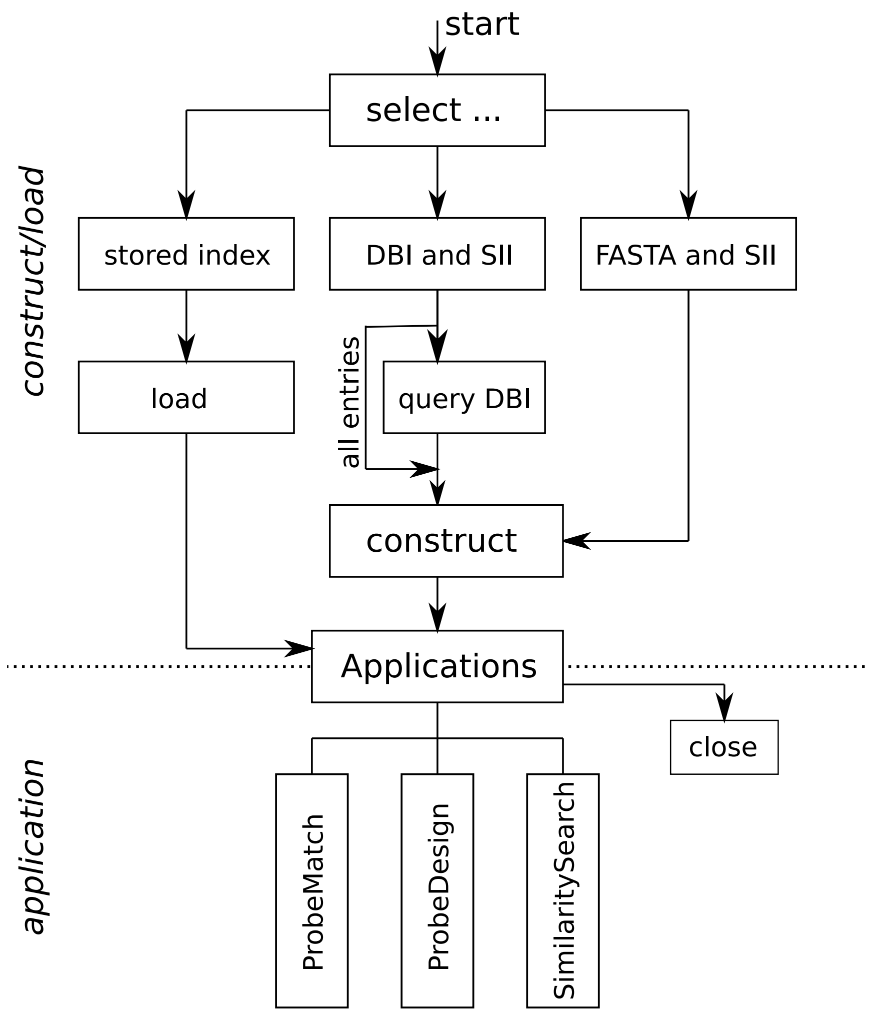


Figure 3.8: *UMDA Primer/Probe (UPP) Designer* workflow overview

4 Evaluation

The next sections provide selected evaluation results gained from testing the index construction, applications and the *UPP Designer* with real-life data. The presented key performance indicators provide the necessary information to assess the index performance.

4.1 Experimental setup

4.1.1 Test Systems

For the tests presented different shared memory systems have been chosen to demonstrate the performance on average as well as top tier desktop systems:

lapbode123 is an average desktop system (*HP EliteBook 8530w*) with an Intel Core2Duo T9400 CPU (2.53 GHz, 2 physical cores), 4 GB DDR2-800 main memory and a Fujitsu MHZ2250B hard disk drive (SATA2, 250 GB, 5400rpm, 12ms average seek time)

atbode223 is a top tier desktop system with an Intel Core i7 920 CPU (2.67 GHz, 4 physical cores) on an ASUS P6T SE mainboard and 24 GB DDR2-1066 main memory. As hard disk drive a Seagate Barracuda LP Series SATA ST31000520AS (SATA2, 1000 GB, 5900rpm, 16ms average seek time) was available.

Both systems run Ubuntu 10.04.3 LTS 64 bit Linux (kernel version 2.6.32). The test compiler was gcc 4.4.3. The boost libraries were utilized in version 1.40.

In addition for some tests the solid state disk model Crucial M4-CT256M4SSD2 SSD (SATA 6G, 256 GB, average access time lower 0.1ms) was available in an eSATA case.

For some distributed memory tests the following system was available:

LRR cluster is a custom Linux cluster build of 24 operational AMD Opteron 4-way nodes, i.e. a total of 96 processors. Each processor is an AMD K8 model 850 single core (SledgeHammer, 2.4 Ghz). Each node provides 8 GB DDR-400 main memory for its 4 processors. The cluster runs under Ubuntu 8.04 64 bit (kernel 2.6.24) with gcc version 4.2.4 and OpenMPI version 1.3.0. The storage system is a 3.4 TB raid system accessible by the network file system over Gigabit Ethernet.

4.1.2 Test Data

Real life gene and genome test data sets have been chosen for measuring the overall construction as well as the application times.

Gene data As large example gene database the SSURef 108 SILVA database (referred to as *SSURef108* further on) was selected [113]. It is a successive database of the SILVA SSURef 104 database chosen for prior tests [32]. The *SSURef108* contains more than 618 thousand sequence entries summing up to a total of more than 890 million bases. Further test databases have been generated containing subsets of the complete *SSURef108* database (table 4.1).

database	number of entries	number of nucleotides
SSURef108	618 442	890 244 367
SSURef104	512 037	738 883 451
400000	400 000	577 172 089
300000	300 000	432 867 080
200000	200 000	288 562 544
100000	100 000	144 349 921
50000	50 000	72 184 577
20000	20 000	28 824 132
10000	10 000	14 414 461
5000	5 000	7 219 680
2000	2 000	2 884 537
1000	1 000	1 446 632

Table 4.1: *SSURef108* gene database and subsets characteristics

database	number of entries	number of nucleotides
GR130	1 499	4 816 402 998
1000	1 000	3 207 188 199
500	500	1 596 164 902
200	200	624 133 762
100	100	301 861 046
50	50	164 142 352
20	20	67 285 904
10	10	33 720 813

Table 4.2: *GR130* genome database and subsets characteristics

Genome data In order to test the *PTPan* index as well as the *UPP Designer* capabilities for genome sequence data, a genome database has been generated in the ARBDB format based on the *GenomeReviews release 130* available as MySQL database dump and as flat file export [41]. The extracted genome database (referred to as *GR130* in the following) contains the cellular prokaryote genomes with annotation data as well as a phylogenetic tree based on the one incorporated in the SILVA SSURef 108 database (database preparation done by H. Meier). The *GR130* database contains 1499 genome sequence entries summing up to a total of 4.8 billion bases. Furthermore for some tests an export in the *multiFASTA* file format containing the bare sequences was used. Based on *GR130* several subdatabases have been generated (table 4.2).

For further tests the MySQL dump of the Genome Reviews release 130 has been utilized (*MySQL-GR130*). It is based on the EnsEMBL database schema (refer to section 2.1.3) and provides a comprehensive collection of genomes along with their annotation data.

4.2 PTPan Index

In the next sections the sequential and parallel *PTPan* construction times for the different test databases are presented. The parallel construction times are divided into shared and distributed memory tests. In addition the influence of the ratio factor is presented. Finally the index sizes are analyzed.

4.2.1 Sequential Construction

The sequential construction times have been measured on the systems *lapbode123* and *atbode223* for *PTPan*. As done before in an evaluation based on SILVA SSURef 104 [32], the ARB PT-Server construction times have been measured for comparison on both systems as well.

SSURef108 On *atbode223* both *PTPan* and the PT-Server could be constructed for all test databases. The construction times are almost equal up to the 300 000 sequence entries sub-database (460 million bases). Afterwards *PTPan* was constructed faster than the PT-Server. For the complete *SSURef108* a construction time of less than 24 minutes for 21 partitions was observed for *PTPan* while the PT-Server required almost 32 minutes to finish. The cause is an alteration of the PT-Server construction algorithm if reaching a memory dependent sequence bases threshold. This is done in order to circumvent more disadvantageous memory effects (R. Westram, ARB maintainer, personal communication).

On *lapbode123*, *PTPan* was constructed up to the full *SSURef108* in about 53 minutes for a total of 85 partitions. Compared to *atbode223* the significantly increased times are caused by the limited memory size. Furthermore, as the ARB database *SSURef108* requires more than 4 GB main memory, the last run was conducted with a *multiFASTA* file export of the *SSURef108* database. In contrast, the ARB PT-Server could be build efficiently only up to the 200 000 sequence entries database. For the larger databases the construction process started to use swap memory and could not finish within reasonable time.

All construction time values for *PTPan* and the PT-Server as well as the number of partitions for *PTPan* are presented in table 4.3. Figure 4.1 prints a direct comparison of the *PTPan* and PT-Server construction times for an increasing amount of nucleotide bases based on the *SSURef108* test database and its subsets.

GR130 In order to compare the performance of *PTPan* and the ARB PT-Server for genome databases, the *GR130* test database construction times have been measured.

PTPan could be constructed for all genome test databases on *atbode223*. For the full *GR130* this took 5 hours 53 minutes resulting in 85 partitions. The PT-Server could only be constructed up to the 500 genomes test database (about 1.6 billion bases) for the same reasons as for the *SSURef108*, i.e. the construction algorithm starts utilizing swap memory on secondary storage. This slows down the construction significantly preventing it from finishing in reasonable time.

On *lapbode123 PTPan* could be constructed up to the 200 genome entries database in about 53 minutes for 84 partitions. The full *GR130* was constructed from a *multiFASTA* file on *lapbode123* taking little less than 22 hours for 448 partitions. In contrast, the ARB

database	*PTPan* (l)	*PTPan* (a)	PT-Server (l)	PT-Server (a)
SSURef108	*53.10 (85)	23.42 (21)	–	31.67
SSURef104	47.80 (85)	20.51 (21)	–	30.69
400000	37.14 (85)	13.81 (9)	–	23.74
300000	23.12 (65)	9.85 (5)	41.85	12.18
200000	9.73 (25)	6.55 (5)	28.25	7.96
100000	4.63 (21)	3.33 (1)	6.78	4.13
50000	2.04 (9)	1.64 (1)	3.25	1.90
20000	0.78 (5)	0.65 (1)	0.87	0.74
10000	0.39 (1)	0.33 (1)	0.40	0.37
5000	0.20 (1)	0.17 (1)	0.20	0.18
2000	0.09 (1)	0.07 (1)	0.08	0.07
1000	0.04 (1)	0.04 (1)	0.04	0.04

Table 4.3: *SSURef108* sequential construction times [min] for *PTPan* (with number of partitions) and PT-Server (a = atbode223, l = lapbode123) *[*multiFASTA* file as source]

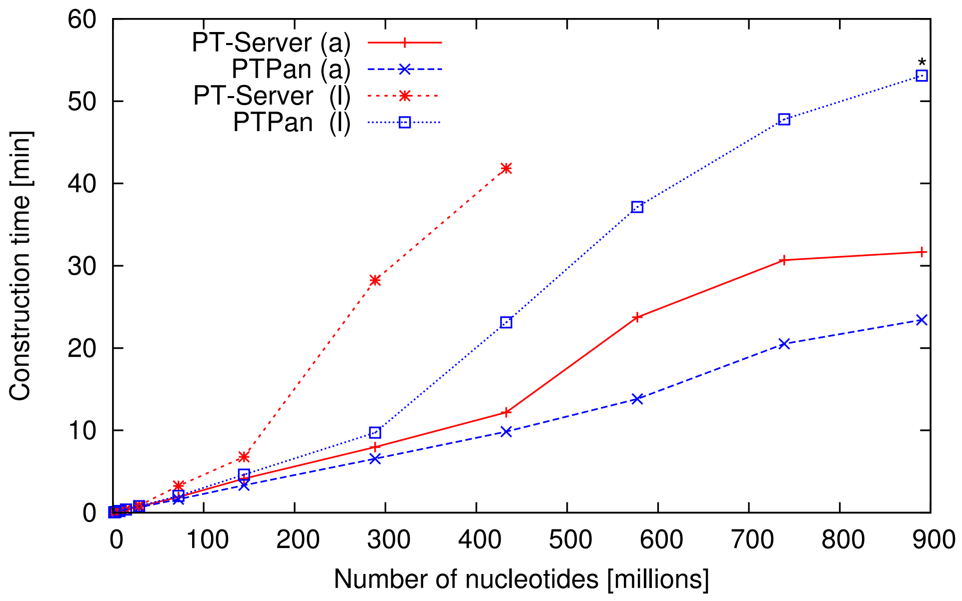


Figure 4.1: *SSURef108* sequential construction times (a = atbode223, l = lapbode123) *[*multiFASTA* file as source]

PT-Server could be constructed only up to the 100 genome test database in reasonable time. Up to this database, the construction times of PT-Server and *PTPan* are comparable, slightly favoring the PT-Server.

All construction time values for the genome test database *GR130* and its subsets as well as the number of partitions for *PTPan* are presented in table 4.4. Figure 4.2 shows the direct comparison of *PTPan* and PT-Server construction times on both test systems.

database	*PTPan* (l)	*PTPan* (a)	PT-Server (l)	PT-Server (a)
GR130	*[1312.40 (448)]	353.00 (85)	-	-
1000	- (-)	227.35 (81)	-	-
500	- (-)	78.64 (21)	-	87.13
200	52.89 (84)	28.75 (13)	-	26.62
100	17.43 (29)	12.90 (5)	15.56	10.31
50	8.64 (21)	6.25 (5)	7.19	5.29
20	3.23 (5)	2.52 (1)	2.41	2.12
10	1.55 (4)	1.28 (1)	1.13	0.87

Table 4.4: *GR130* sequential construction times [min] for *PTPan* (with number of partitions) and PT-Server (a = atbode223, l = lapbode123) *[*multiFASTA* file as source]

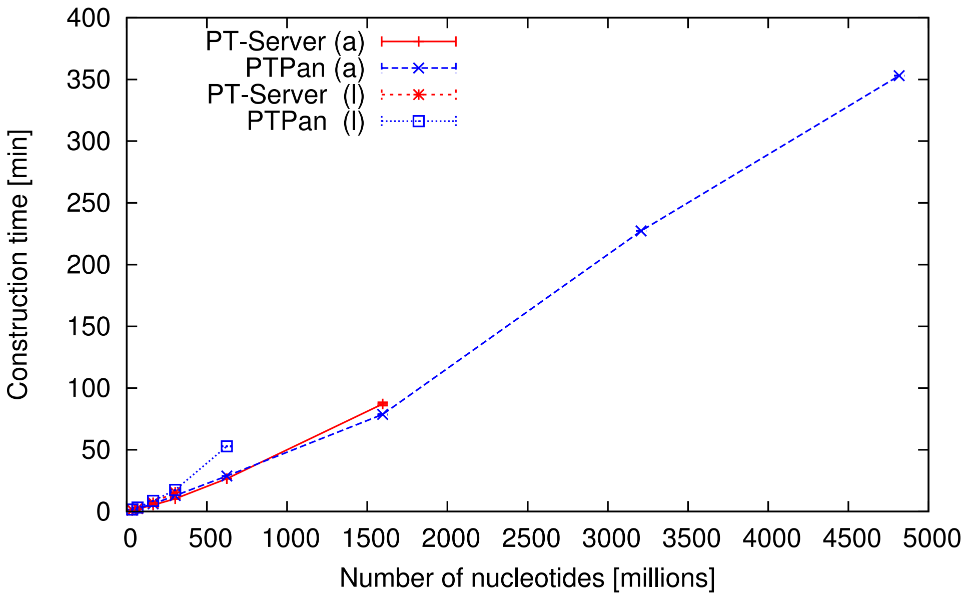


Figure 4.2: *GR130* sequential construction times (a = atbode223, l = lapbode123)

4.2.2 Parallel Construction

4.2.2.1 Shared Memory

In order to test the parallel *PTPan* construction algorithm, runtime tests have been conducted for the shared memory test systems up to their number of physical cores, i.e. four for *atbode223* and two for *lapbode123*.

SSURef108 For *lapbode123* the construction time for one thread is 53 minutes for 85 partitions which is better compared to the 97 minutes for 248 partitions with two threads. The partition determination with automatic optimal thread count selection (refer to section 3.4.1.2) chooses the best value in this case, i.e. one thread with 85 partitions.

For *atbode223* the sequential construction time was about 24 minutes for 21 partitions. Utilizing two, three or four threads results in runtime values ranging in between 13 and 14 minutes for 21, 53 respectively 73 partitions. The speedup factors compared to sequential construction time for two to four threads are nearly identical ranging from 1.67 to 1.8. The best runtime and time-to-partitions ratio is achieved with two threads.

Table 4.5 shows all parallel construction time values for *SSURef108* on both systems.

system	threads	partitions	time [min]	speedup
lapbode123	**1**	**85**	**53.1**	**1.00**
lapbode123	2	248	97.4	0.55
atbode223	1	21	23.42	1.00
atbode223	**2**	**21**	**13.03**	**1.80**
atbode223	3	53	14.05	1.67
atbode223	4	73	13.33	1.76

Table 4.5: *SSURef108* parallel construction times [bold = automatically chosen]

GR130 The parallel construction tests for *GR130* have been conducted on *atbode223* only. The sequential runtime is 353 minutes for 85 partitions, also offering the best threads-to-partitions ratio. In comparison the parallel construction time for four threads is the lowest with 248 minutes for 345 partitions. Hence the automatic algorithm would choose one thread while the optimal thread count is four. Due to the increasing number of partitions, the overall speedups are low. Utilizing four threads results only in a maximum speedup of 1.42. Table 4.6 shows all values for parallel *GR130* construction on *atbode223*.

system	threads	partitions	time [min]	speedup
atbode223	**1**	**85**	**353**	**1.00**
atbode223	2	190	312	1.13
atbode223	3	306	294	1.20
atbode223	4	345	248	1.42

Table 4.6: *GR130* parallel construction times [bold = automatically chosen]

4.2.2.2 Distributed Memory

Construction runtime tests on the *LRR Cluster* have been conducted for both *SSURef108* and *GR130*. With main memory being limited to 2 GB per core, the number of partitions is fixed regardless the number of cores participating. For *SSURef108* 205 and for *GR130* 982 partitions are constructed.

Table 4.7 lists the construction times and speedup factors for an increasing number of cores from 12 up to 96. Construction time speedups are presented for direct comparison in figure 4.3 with the time for 12 cores as reference. The runtime for the *GR130* database on 12 nodes is 766 minutes (about 12.8 hours), declining to 108 minutes (1.8 hours) on 96 nodes. For the *SSURef108* runtime declines from 33.5 minutes (12 nodes) to 7.2 minutes (96 nodes).

In the *MPI PTPan* construction algorithm, one master node is responsible for data retrieval and preparation while the other nodes are slaves responsible for index construction. Having one master regardless of the overall number of nodes participating allows super-linear speedups when increasing the number of nodes. For example the 7-fold increase of participating nodes from 12 to 96 for *GR130* results in a speedup of 7.1.

database	#partitions	12	24	48	96
GR130	982	766 (1.00)	374 (2.05)	191 (4.01)	108 (7.09)
SSURef108	205	33.5 (1.00)	16.9 (1.98)	10.3 (3.25)	7.2 (4.65)

Table 4.7: *SSURef108* and *GR130 PTPan* construction times [min] and speedups on *LRR cluster*

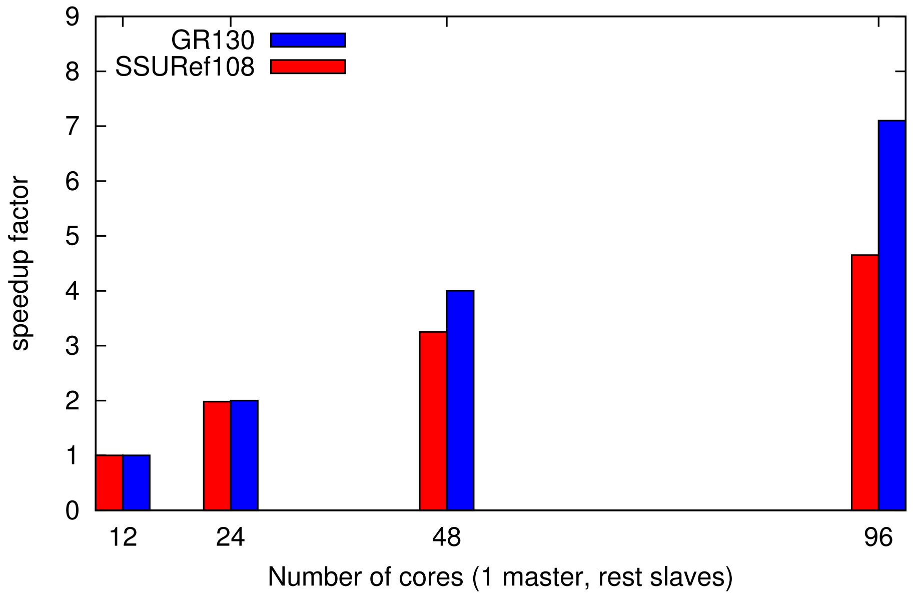


Figure 4.3: *SSURef108* and *GR130 PTPan* construction speedup factors comparison on *LRR cluster*

4.2.3 Ratio Factor Influence

In order to test the influence of a varying memory ratio factor (refer to section 3.4.1), several tests have been performed on the shared memory systems *atbode223* and *lapbode123*.

According to the statistical data gathered during index construction, the worst case estimation of memory requirements is not optimal in all cases. For the complete *SSURef108*, the total memory used was less than 20 percent of the estimated value. For the complete *GR130*, a capacity utilization of 80 percent at maximum was determined.

With these facts and the former observation that an increase in the number of partitions triggers an increased construction time, the ratio factor is expected to lower the runtime of sequential and parallel construction.

SSURef108 For *SSURef108* the ratio factor influence has been investigated by conducting measurements on *atbode223* and *lapbode123* for each combination of hardware thread count and ratio factors 0.5 and 0.2. The results are shown in table 4.8 together with the sequential runtime for the default memory requirements estimation.

For *lapbode123*, utilizing the ratio factors, the number of partitions to construct declined heavily from 85 to 21 at best. With this the construction time has been lowered to less than 25 minutes. This is significantly less compared to 53 minutes as fastest parallel construction runtime for the default main memory requirements estimation. The best speedup of more than factor 2 has been observed for the ratio factor 0.2 with one or two threads. The automatic thread-count choosing algorithm would pick the slightly faster option with one thread for 21 partitions. For factor 0.5 the best option is also correctly selected automatically, i.e. two threads constructing 89 partitions in less than 36 minutes which is still a speedup by factor 1.5.

For *atbode223* the results are similar. The number of partitions for sequential and parallel construction runs declines compared to the default memory requirements estimation when the ratio factors are applied. With this, the runtime drops to a little more than 8 minutes for 21 partitions with factor 0.5 respectively about 7.4 minutes for 9 partitions with factor 0.2, both with four threads. The speedup factors are for both ratios higher compared to the fastest speedup so far, i.e. speedup 2.85 for ratio 0.5 respectively 3.18 for ratio 0.2 compared to a speedup of 1.8 for default settings. The thread-count selection algorithm would pick three threads in the case of a ratio of 0.2 which is not the best choice regarding the overall runtime. Optimal would be a thread count of four. For a ratio of 0.5 it is even worse. One thread and its runtime of about 19 minutes is chosen while the best choice would be to rely on all four hardware threads available in the system. This would result in a runtime of a little more than 8 minutes.

GR130 For the *GR130* database measurements have been conducted on *atbode223* for all different hardware thread counts possible in conjunction with a ratio factor of 0.8. The results are shown in table 4.9 together with the sequential as well as the best parallel runtime for the default memory requirements estimation.

The construction time and the number of partitions are equal for one thread with and without the ratio factor, i.e. 354 minutes for 85 partitions. For multiple threads the number of partitions to construct declines and with it the runtime drops to less than four hours. Regarding the automatic thread count selection algorithm, it chooses 2 threads which indeed

system	ratio	threads	partitions	time [min]	speedup
lapbode123	*1.0*	*1*	*85*	*53.10*	*1.00*
lapbode123	0.5	1	61	42.27	1.26
lapbode123	**0.5**	**2**	**89**	**35.83**	**1.48**
lapbode123	**0.2**	**1**	**21**	**24.38**	**2.18**
lapbode123	0.2	2	45	24.62	2.16
atbode223	*1.0*	*1*	*21*	*23.42*	*1.00*
atbode223	*1.0*	*2*	*21*	*13.03*	*1.80*
atbode223	**0.5**	**1**	**5**	**18.57**	**1.26**
atbode223	0.5	2	17	12.45	1.88
atbode223	0.5	3	21	9.87	2.37
atbode223	0.5	4	21	8.22	2.85
atbode223	0.2	1	5	18.57	1.26
atbode223	0.2	2	5	10.87	2.15
atbode223	**0.2**	**3**	**5**	**9.93**	**2.36**
atbode223	0.2	4	9	7.37	3.18

Table 4.8: *SSURef108* sequential and parallel construction times for different ratio factors [bold = automatically chosen; italic = reference values]

provides the best overall runtime with 216 minutes for 113 partitions. This is a speedup of 1.64 compared to the sequential construction time.

system	ratio	threads	partitions	time [min]	speedup
atbode223	*1.0*	*1*	*85*	*353*	*1.00*
atbode223	*1.0*	*4*	*345*	*248*	*1.42*
atbode223	0.8	1	85	354	1.00
atbode223	**0.8**	**2**	**113**	**216**	**1.64**
atbode223	0.8	3	268	268	1.32
atbode223	0.8	4	306	225	1.57

Table 4.9: *GR130* sequential and parallel construction times with ratio factor [bold = automatically chosen; italic = reference values]

4.2.4 UMDA Based Construction

In order to test the capabilities of building an index with the *UPP Designer* (refer to section 3.5), the *PTPan SII* has been utilized in conjunction with the *EnsEMBL DBI* plugin connecting to the *MySQL-GR130* test database.

The database has been queried for subsets of genome sequence entries with help of the *UMDA DBI QueryAPI*. The result entries have been taken to construct an index.

Profiling revealed that the overall *PTPan* index construction time is dominated by the construction of the partitions. The data retrieval contributes only to a lower amount. Thus the construction runtime results presented in the previous sections can be conferred to other data input sources presented in this work as well. Hence the construction time results for the different subdatabase selections are not shown explicitly.

4.2.5 Index Size

The index size on disk was analyzed for *SSURef108* and *GR130*. As it proved to be independent of the number of partitions, only the values for the indexes constructed on *atbode223* with the sequential algorithm are presented.

For comparison to *PTPan*, the values for the ARB PT-Server are presented as well. As the ARB PT-Server relies on the original sequence data source for application, the summarized values of index and database size are presented as well.

PTPan relies on loading only required parts of the index into main memory. Hence the on-disk memory requirements are not equal to the main memory requirements.

In contrast, the ARB PT-Server must fit into main memory completely for application. As it is stored compressed, the on-disk size is a rough lower limit for the main memory requirements.

database	*PTPan*	PT-Server
SSURef108	3 413	3 236 (3 922)
SSURef104	3 039	2 790 (3 337)
400000	2 395	2 208 (2 639)
300000	1 818	1 682 (2 010)
200000	1 233	1 148 (1 372)
100000	636	596 (716)
50000	332	310 (379)
20000	141	133 (170)
10000	75	71 (99)
5000	40	39 (60)
2000	17	17 (53)
1000	10	10 (37)

Table 4.10: *SSURef108 PTPan* and ARB PT-Server stored index sizes [MB] (in brackets incl. database size)

database	*PTPan*	PT-Server
GR130	39 340	-
1000	27 047	-
500	14 042	-
200	5 907	8 298 (8 963)
100	2 838	4 016 (4 342)
50	1 521	2 209 (2 386)
20	668	907 (982)
10	352	452 (487)

Table 4.11: *GR130 PTPan* and ARB PT-Server stored index sizes [MB] (in brackets incl. database size)

SSURef108 The index sizes for both *PTPan* and PT-Server are presented in table 4.10. Looking at the index only, the sizes are nearly identical for *PTPan* and PT-Server, although *PTPan* includes all data required for application, comprising the original sequence and its related data. For the complete *SSURef108* with its 890 million nucleotides, the *PTPan* index is about 3400 MB in size which is less than $4N$ bytes per nucleotide.

GR130 In table 4.11 the results for the genome databases are presented for both *PTPan* and PT-Server. As it can be seen, *PTPan* has reduced requirements for genome data compared to the PT-Server index size. The overall index size of *PTPan* for the complete *GR130* with its 4.8 billion nucleotides is slightly less than 40 GB, which is about $8.6N$ bytes per base.

4.2.6 Summary

The *PTPan* index construction capabilities have been evaluated for the sequential algorithm as well as the parallel algorithms for shared and distributed memory systems.

The sequential construction times for the real life gene and genome data sets revealed *PTPan* outperforming its competitor, the ARB PT-Server. In addition, *PTPan* is capable of dealing with large amounts of sequence data even on computer systems with limited main memory capacity while the ARB PT-Server reaches its limitations.

The parallel *PTPan* construction algorithms proofed to reduce the construction times. However, the increasing amount of partitions to construct when utilizing more than one computing core on a multi-core system sometimes negate the benefits of parallel construction. The automatic determination algorithm estimating the best thread-count-to-partition ratio proofed to ease this problem. It selects the best combination most of the times.

Furthermore the ratio factor parameter, which must be provided manually based on empiric data, showed its potential to reduce construction times significantly for both kinds of test data, i.e. gene and genome sequences.

On distributed memory systems, the MPI based *PTPan* construction algorithm scales well for an increasing amount of participating computing nodes. With this, the overall *PTPan* construction time for large amounts of data has been reduced significantly.

Further tests revealed that the construction time is dominated by the construction of the partitions. The data retrieval contributes only to a smaller extend. With this, the type of the sequence data source, for example a relational database or the in-memory ARB database, does not influence the overall construction time significantly for large data collections.

Finally the *PTPan* index size increases linear to the amount of gene or genome sequence data the index is based on. *PTPan* indexes require less size in sum compared to ARB PT-Server ones. The latter require that the original source ARB database is available during application.

4.3 Applications

In the next sections the evaluation of the primer/probe design and evaluation applications will be presented. First the *ProbeMatch* application and its runtime behavior is shown. This is followed by presenting mass query tests based on the *UPP Designer* accessing *ProbeDesign* and *SimilaritySearch* utilizing the *PTPan SII* plugin.

4.3.1 PTPan ProbeMatch

PTPan ProbeMatch has been tested utilizing the *PTPan* ARB integration (refer to section 3.5). This allows to directly compare the results to the ones obtained for the ARB PT-Server. All measurements are based on the example probe *EUB338*, a sequence well known to have a very high number of entry hits in SILVA databases [3]. As all genomes in *GR130* are represented in *SSURef108* by their corresponding extracted genes, the *EUB338* should provide a high number of hits in *GR130* as well.

All presented tests have been conducted in *basic-match-mode* (refer to section 3.3.2) for exact as well as Levenshtein-distance metric based approximate queries up to five allowed mismatches. Test systems were the two shared memory systems *atbode223* and *lapbode123* utilizing their internal hard disk drive. For each test query, the index is loaded freshly to prevent the influence of caching effects provided by the memory management of the operating system.

For the *EUB338* single match queries the runtime utilizing the SSD drive did not differ significantly from the normal disk drive values. Thus they are not shown explicitly.

SSURef108 The number of hits delivered by *PTPan ProbeMatch* for the complete *SSURef108* are shown in table 4.12. For distances from 0 to 3 the count grows moderately while it starts increasing faster for distances of 4 and 5.

database	max distance	hits
SSURef108	0	487 342
SSURef108	1	509 681
SSURef108	2	524 086
SSURef108	3	553 509
SSURef108	4	689 445
SSURef108	5	1 215 486

Table 4.12: *SSURef108 ProbeMatch* query hits for *EUB338*

This increase of hit numbers influences the overall runtime as well (table 4.13). For *atbode223* sequential query times range from 7.3 seconds for 0 mismatches up to 13.9 seconds for a maximum distance of 4. *lapbode123* query times are slower ranging from 8.1 to 15.6 seconds. Utilizing more than one thread decreases the runtime for all tested distances on both test systems. For direct comparison, the sequential and parallel query times are shown in figure 4.4 for *atbode223* and in figure 4.5 for *lapbode123*. By this a similar overall runtime tendency for increasing distance values on both systems is revealed.

system	threads	max distance					
		0	1	2	3	4	5
lapbode123	1	8.1	8.7	9.0	10.0	15.6	86.9
lapbode123	2	5.7	6.1	6.3	6.9	14.1	77.5
atbode223	1	7.3	7.8	8.0	8.7	13.9	52.1
atbode223	2	4.6	5.2	5.2	5.8	9.6	38.6
atbode223	4	3.4	3.8	4.0	4.4	7.3	32.4

Table 4.13: *SSURef108 ProbeMatch* query times [sec] for *EUB338*

In figure 4.6 the speedup factors are presented. The slowest runtime functions as reference, i.e. the one on *lapbode123* with one thread. As it can be seen, the runtimes for 0 to 4 allowed mismatches differ only slightly for one thread on both systems. In contrast, the parallel runs are in favor of *atbode223* providing higher speedups. This can be explained by the faster CPU and the larger main memory. On *lapbode123*, after conducting the pattern matching, for building the differential alignment it is necessary to free memory in order to be able to load the original sequence data into main memory. On *atbode223* this is not a big issue due to the large main memory size.

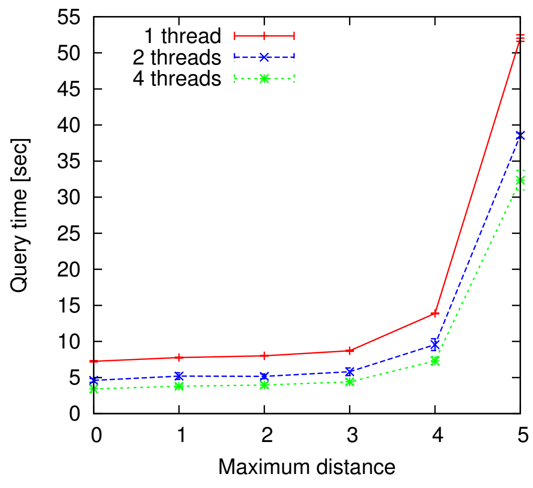


Figure 4.4: *SSURef108 ProbeMatch* query times on *atbode223* for *EUB338*

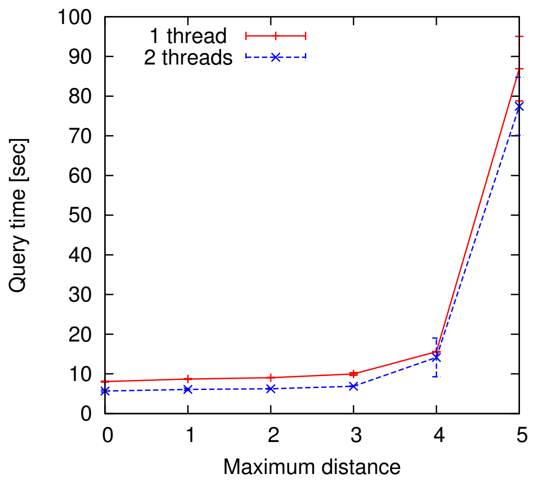


Figure 4.5: *SSURef108 ProbeMatch* query times on *lapbode123* for *EUB338*

For comparison the ARB PT-Server has been tested for the complete *SSURef108* on *atbode223* as well (table 4.14). The results shown reveal query times of 2 to 3 seconds depending on the number of mismatches allowed. The overall hit numbers for zero mismatches are almost identical for *PTPan* and the PT-Server. For 1 to 5 mismatches allowed, they start to drift apart. This is caused by the fact that the ARB PT-Server relies on the Hamming-distance metric to identify substitutions. It can not find hits with insertions or deletions (indels) as *PTPan* utilizing the Levenshtein-distance metric does.

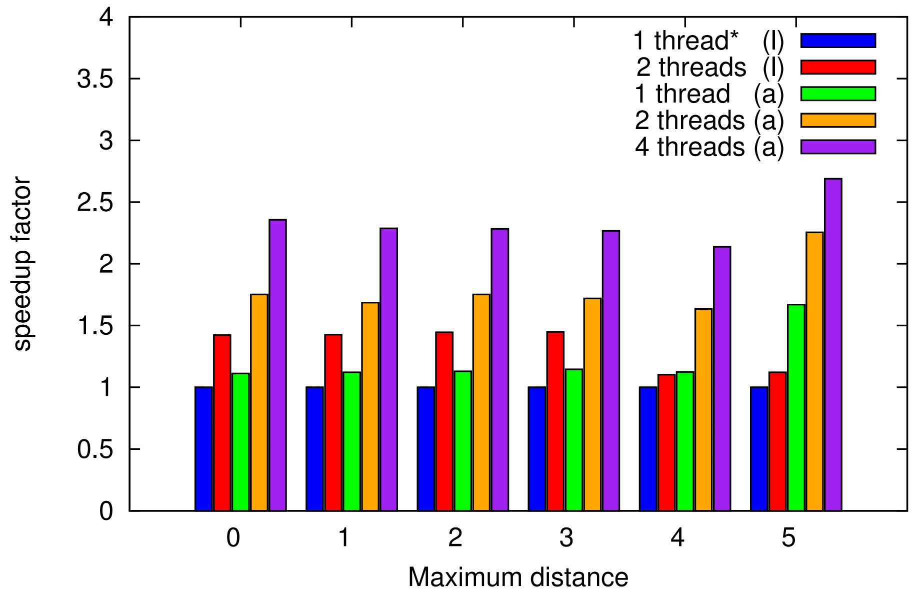


Figure 4.6: *SSURef108 ProbeMatch* query time speedups for *EUB338* (l = lapbode123; a = atbode223; *reference value)

The gap in query times between *PTPan* and the PT-Server is caused by *PTPan* requiring to decompress the index during application. In addition *PTPan* identifies more hits in contrast to the PT-Server as indels are spotted as well. Compared to the previous evaluation presented in [32], the query times gap between *PTPan* and the PT-Server has been narrowed. Instead of slowdown factors from 4 to 15 for 0 to 4 mismatches, it was lowered to about 4 to 6.5 times comparing the sequential query times observed on *lapbode123*. For *atbode223* with four threads, the query time gap even narrowed further down to a range of 1.7 to 3 times slowdown.

max distance	time	hits
0	2.04	487 207
1	2.19	508 183
2	2.53	522 100
3	2.26	532 341
4	2.44	570 920
5	2.91	666 246

Table 4.14: *SSURef108* ARB PT-Server query hits and times [sec] for *EUB338*

GR130 The *ProbeMatch* runtime tests for *GR130* draw a similar picture to the one presented for *SSURef108*. The number of hits delivered by *PTPan ProbeMatch* are shown in table 4.15. Again, for distances from 0 to 3 the count grows moderately while it starts increasing faster for distances of 4 and 5.

database	max distance	hits
GR130	0	2 967
GR130	1	3 035
GR130	2	3 203
GR130	3	8 719
GR130	4	118 771
GR130	5	1 527 577

Table 4.15: *GR130 ProbeMatch* query hits for *EUB338*

This increase of hit numbers influences the overall runtime as well (table 4.16). For *atbode223* and *lapbode123* sequential query times are almost identical in the range from 1.1 seconds for 0 mismatches up to about 3.3 seconds for a maximum distance of 3. Starting with 4 mismatches allowed, the sequential query times start to increase significantly but are still reasonable. The query times for 5 mismatches are by far larger than the other ones and can be considered impractical for certain applications as they lie in the range of several minutes.

The sequential and parallel runtimes are shown in figure 4.7 for *atbode223* and in figure 4.8 for *lapbode123* revealing a similar overall runtime tendency for increasing distance values on both systems.

system	threads	max distance					
		0	1	2	3	4	5
lapbode123	1	1.10	1.07	1.20	3.29	67.31	1113.43
lapbode123	2	0.67	0.68	0.72	2.00	37.40	1052.53
atbode223	1	1.12	1.14	1.6	3.20	41.00	533.44
atbode223	2	0.68	0.65	0.73	1.90	22.64	304.87
atbode223	4	0.60	0.64	0.63	1.51	15.88	181.82

Table 4.16: *GR130 ProbeMatch* query times [sec] for *EUB338*

In figure 4.9 the speedup factors are presented with the runtime for one thread on *lapbode123* being the reference. As it can be seen, the runtimes for 0 to 3 allowed mismatches differ only slightly for one thread on both systems. For a maximum distance of 4 and 5, *atbode223* has a clear advantage. Compared to the low error rate queries, the higher the allowed error rate, the more parts of the index are loaded into main memory. As for the *SSURef108* tests, *atbode223* with its 24 GB main memory does not require to free main memory early. It can keep large parts of the index. The *lapbode123* in contrast has to free main memory which results in a higher overall runtime. The parallel runs are also in favor of *atbode223*, although they do not differ much between two and four threads for 0 to 3 maximum distance. In contrast for a maximum distance of 4 and 5, the speedup factors for four threads are the highest. The *lapbode* speedup factors with the second thread are nearly identical at about 1.6 up to 4 allowed mismatches.

The tests for the ARB PT-Server could not be conducted as it was not possible to con-

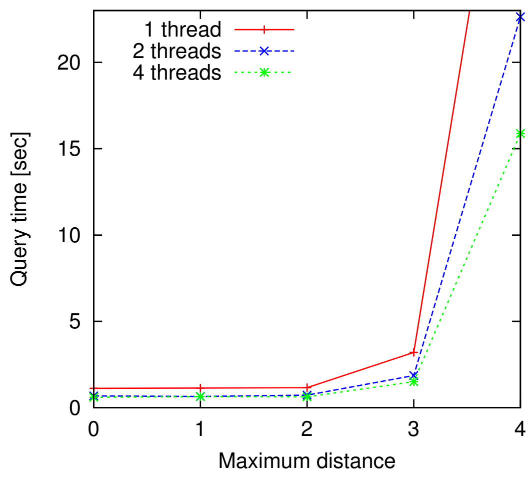


Figure 4.7: *GR130 ProbeMatch* query times on *atbode223* for *EUB338*

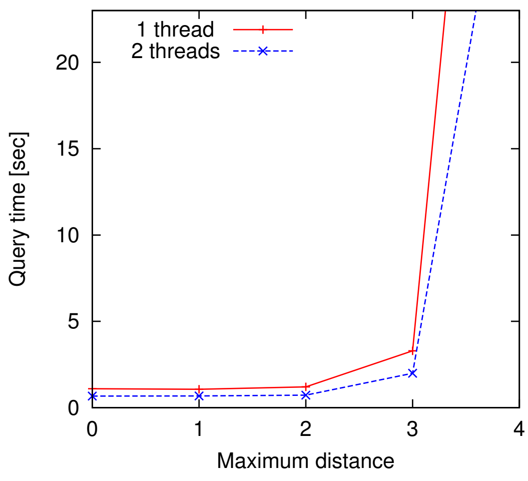


Figure 4.8: *GR130 ProbeMatch* query times on *lapbode123* for *EUB338*

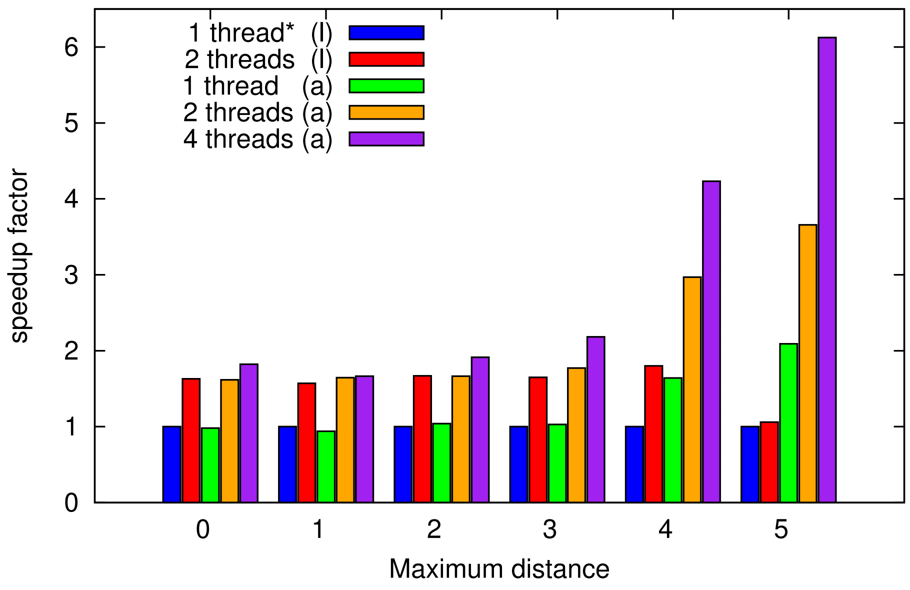


Figure 4.9: *GR130 ProbeMatch* query time speedups for *EUB338* (l = lapbode123; a = atbode223; *reference value)

struct it for the complete *GR130* database even on *atbode223* with its large main memory (refer to section 4.2.1).

Comparison To BWA The preliminary version of *PTPan* presented in [32] has been compared to the short read mapper *BWA* [77] (refer to section 2.4.4.4). Test database has been the SILVA release SSURef 104 gene database. Compared to *PTPan BWA* showed faster construction times and lower memory demands at the price of increased approximate string matching times. Furthermore it is not capable of providing all demanded information for primer/probe design as it does handle only the *DNA4* alphabet and not the *DNA5* like *PTPan* does. Furthermore a differential alignment of the single hits is available in the CIGAR format only which does not show the exact substituted base (refer to section 2.1.2). A weighted mismatch scheme is not supported either.

The tests for gene databases conducted in [32] have not been repeated with *SSURef108*. Instead, in order to see if *BWA* behaves similar with genome data, query tests have been conducted based on the *GR130 multiFASTA* file with *BWA version 0.6.1r104* on *atbode223*.

The *BWA* index was constructed in 118 minutes which is nearly half the time compared to *PTPan*s best construction time of 216 minutes with two threads and a ratio factor of 0.8 (refer to section 4.2.3). The final *BWA* index requires about 8 GB memory on disk.

After construction, the index has been queried for *EUB338* the same way described in [32], i.e. with the settings adopted to return all matches up to the predefined distance, not only the best one. The query times returned as output by BWA itself are shown in table 4.17 along with the hit numbers. The query times are about 6 times slower compared to *PTPan* for maximum allowed distance ranging from 0 to 3.

max distance	time [sec]	hits
0	6.48	5 301
1	6.53	16 008
2	6.79	32 590
3	18.40	68 702
4	6.90	49 722
5	6.92	49 722

Table 4.17: *GR130 BWA* query times and hit numbers for *EUB338*

For higher error rates, the behavior is the same as in the former evaluation [32]. The query times and hit numbers drop suddenly. Still no adequate explanation for this behavior is available as the query for 4 and 5 allowed mismatches should theoretically include the matches from the 3 mismatches query as well. The higher hit numbers compared to *PTPan* for the lower error rates can be explained by the fact that *BWA* includes an automatically query on the reverse sequence as well with no way to turn this behavior off. But even dividing the query times by factor two would result in higher values.

4.3.2 UPP Designer Based Mass Queries

Besides the single query capabilities of *ProbeMatch*, the mass query capabilities in form of *ProbeDesign* and *SimilartitySearch* have been tested. The tests have been conducted for the *GR130* based *PTPan* index on *lapbode123* and *atbode223* with the internal hard disk drives as well as the SSD drive (refer to section 4.1.1). As test software the *UPP Designer* and the *PTPan SII* plugin were utilized (refer to sections 3.5.4 and 3.5.3).

SII based ProbeDesign For *ProbeDesign* (refer to section 3.3.3) two different tests have been carried out based on the target group of all 1398 genome sequence entries of the domain bacteria contained in *GR130*.

First only the seek time has been measured, i.e. the time to obtain all signature candidates omitting the subsequent parts of the algorithm, namely the *ProbeMatch* and the quality calculation. The times measured are about 36 minutes on *lapbode123* respectively 25 minutes on *atbode223* (table 4.18). A difference between the internal hard disk drives and the external SSD has not been observed. This shows that the index performs well on common HDDs it has been intended for in the first place as well as on modern SSDs independently of the available main memory size. Furthermore it reflects the same behavior concerning index access times for a freshly loaded index as observed in the single *ProbeMatch* query tests (refer to section 4.3.1).

system	disk drive	seek [sec]	complete [sec]
lapbode123	HDD	2 166	4 448
lapbode123	SSD	2 134	2 725
atbode223	HDD	1 496	2 511
atbode223	SSD	1 503	1 747

Table 4.18: *GR130 ProbeDesign* times for seek only and the complete algorithm

As second test, the complete *ProbeDesign* process has been carried out including the subsequent *ProbeMatch* and quality calculation for the seeked candidate list. A significant difference for the complete *ProbeDesign* runtimes was determined on both test systems. Relying on SSDs, the overall runtime speedup was 1.44 on *atbode223* and 1.63 on *lapbode123* requiring no changes to the application algorithm or code.

SII based SimilaritySearch In order to further test *PTPan* mass query capabilities, *SimilaritySearch* has been conducted for the genome test database *GR130* on both *atbode223* and *lapbode123*. The selected test results are based on a genome sequence with more than 2.17 million bases resulting in over 2.17 million queries for the complete *SimilaritySearch* process.

On *atbode223* the aggregated query time utilizing the internal HDD is about 55 minutes. Taking the SSD drive it dropped under 21 minutes which is a speedup by factor 2.6. In contrast, the aggregated query times on *lapbode123* with about 7.3 hours for the common HDD are significantly higher compared to *atbode223*. This can be explained by the larger main memory of *atbode223* functioning as cache for parts of the index. But as for *atbode223*,

system	disk drive	time [sec]
lapbode123	HDD	26 022
lapbode123	SSD	3 761
atbode223	HDD	3 297
atbode223	SSD	1 233

Table 4.19: *GR130 SimilaritySearch* times

mass queries on *lapbode123* profit from a SSD drive as well, lowering the overall time to about 63 minutes which is a speedup by almost factor 7. As for the *ProbeDesign* test, no algorithmic or code changes have been made to achieve the speedup with the SSD.

4.3.3 Summary

The primer and probe design and evaluation capabilities of *PTPan* have been tested based on both the *UPP Designer* as well as the ARB integration of *PTPan*.

The results for *PTPan ProbeMatch* revealed the ability of *PTPan* to perform well for real life gene and genome data sets on the various different sized test systems, even when the index is larger than the available main memory. Comparing the gene data query times for *PTPan* and the ARB PT-Server reveals an acceptable slowdown by low factors for *PTPan*. This slowdown is caused by *PTPan* residing on secondary storage, requiring to load and decompress during application, as well as the utilization of the more compute intensive Levenshtein-distance metric for approximate string matching. Furthermore *PTPan* is able to utilize multicore-architectures to speedup the *ProbeMatch* query times.

If conducting mass queries, as it is done by *PTPan ProbeDesign* and *SimilaritySearch*, *PT-Pan* performs well on the different sized test systems. In addition, the utilization of SSDs as secondary storage results in a significant speedup of the overall mass query runtime, especially on systems with limited main memory.

5 Discussion

The increasing amounts of available nucleic acid genome sequence data offers new possibilities for sequence based analysis like computational molecular diagnostics (CMD). CMD is based on the in-silico search for molecular markers, primers and probes based on sequence and phylogenetic data and can lead to a faster development of molecular detection methods for pathogens while reducing the experimental cost in the wet lab.

First the genome data must be accessed and managed efficiently in a unified way and second it must be processed often supported by index structures. New solutions must take care of hardware architecture specifics such as limited main memory and multi-core processors to perform well. In the following sections, the work presented in this thesis is compared to other existing approaches.

5.1 Unified Molecular Data Access Framework

The presented *Unified Molecular Data Access (UMDA)* framework concept (refer to section 3.1) introduced an object model suitable for molecular sequence data and related information as well as analysis results such as primers and probes. In addition generic abstract interfaces for database access (*DBI*) and for search index based primer/probe design and evaluation capabilities (*SII*) have been presented. The *DBI* allows the integration of different heterogeneous molecular sequence databases and has been implemented for different genome database schemas and database management systems (DBMS). The *SII* allows generic access to different index types and has been implemented for *PTPan* (refer to section 3.2) and prototypical based on the Seqan framework [30].

The *UMDA object model* introduced in section 3.1.2 provides a generic and extensible representation of molecular sequence and related data. Comparing it with other existing approaches like *BioJava* [56], *BioPerl* [126], *BioPython* [23] and the *NCBI C++ Toolkit* [103] reveals that some data objects are common for all APIs, for example *SequenceEntry* and *Sequence*. Most of the APIs offer object types for alignments, although other definition and result objects like *Partition* or *APAPM* are *UMDA* specific. Furthermore, the *UMDA common objects Trait* and *QueryBrick* are unique (refer to section 3.1.3). The unified *UMDA object model* facilitates the development of abstract interfaces like the *UMDA DBI* and *SII* in a way not offered by any other project.

The *UMDA DBI*, presented in section 3.1.3.2, abstracts from the underlying DBMS and database schema providing an integrative interface. The abstraction is of central importance for the *AccessAPI* defining a single point of integration for specific databases. Only the programmer of the specific *DBI* needs to know the underlying database schema. This facilitates the usage of a database significantly. Furthermore the *QueryAPI* allows for the flexible, database system and schema independent building of queries. This is achieved by using *QueryBricks* which do not depend on any specific query system or database schema. Available fields for querying can be retrieved together with the appropriate brick type. The bricks are evaluated by the different database interfaces. The latter build a DBMS and schema specific query using the information provided by the bricks.

A concept comparable to the *AccessAPI* is the common database interface provided by the *NCBI C++ Toolkit* [103]. In contrast to *UMDA* it abstracts only from the relational DBMS, but not from the schema used. Hence all queries have to be explicitly designed by the user. *BioJava*, *BioPerl* and *BioPython* do not currently offer an integrative database interface [11, 12, 13]. However all projects offer access to *BioSQL* databases and the established flat file formats. In addition *BioPerl* offers access to EnsEMBL and, with the GMOD modules, to the *Chado* genome database schema as well [21]. Furthermore, according to its documentation, *BioPerl* offers a limited approach for querying *Genbank* by providing a query item with a database-specific argument list [12].

Other sequence data integration approaches are available in form of data warehouses like *BioMart* [122] or database integration systems like *ACNUC* [47] or the *Sequence Retrieval System (SRS)* [36]. All of them integrate data from a large number of publicly accessible biological databases, including but not limited to molecular sequence databases, with the goal of making it available for efficient querying or as read-only data sources. The different systems mostly offer the opportunity for automatic integration and update of the incorporated data. However the mentioned systems have major drawbacks. Unfortunately most system are complicated to use and require advanced hardware equipment as stated by Shah et al. [121]. In addition Töpel and colleagues notice that biological warehouses are focused on the global integration of a wide range of different biological data available and are commonly web-oriented [133].

Contrary to these systems the *UMDA* framework is aimed towards usage in client applications to build scalable solutions on a wide range of different computer hardware. Furthermore it can be utilized to access local databases which is of great interest to build secondary databases with user selected genome sequence entries. The necessary data can be retrieved from primary sources exploiting the *QueryAPI* in conjunction with the *AccessAPI*. In addition this facilitates the inclusion of unpublished results into in-silico analysis.

The *UMDA SII*, presented in section 3.1.3.3, offers a layer of abstraction for search index based primer/probe design and evaluation capabilities. It is a unique feature not present in other applications or APIs so far. The only other Bioinformatics related framework offering search index capabilities is *Seqan* [30, 44]. Although it offers implementations of common index types like the suffix tree and enhanced suffix array in their generic forms only. It does not comprise a complete application package for primer/probe design and evaluation.

5.2 PTPan Index And Applications

The presented *PTPan* index structure (refer to section 3.2) introduced a compressed search index based on a truncated and partitioned suffix tree on secondary storage. The new suffix tree stream-compression employed reduces the memory requirements and enables efficient top-down traversal in depth-first-search (DFS) order. This enables efficient Levenshtein-distance metric based non-heuristic approximate string matching even when the index does not fit into main memory. *PTPan* has been developed and optimized to support similarity searches and primer/probe design in huge nucleic acid sequence collections, highly demanded in molecular microbial diagnostics.

PTPan is based on suffix trees on secondary storage which the review conducted as part of this thesis revealed to be the most appealing approach (refer to section 3.2.1). It was chosen over other approaches utilizing secondary storage as well as over self-indexes like the *CSA* successfully employed by the non-heuristic short read mapper *BWA* aiming at identifying the best hit for a short read [78]. Nevertheless *BWA* has been evaluated thoroughly for gene sequence data as presented in [32]. The evaluation revealed the beneficial low memory demands and moderate construction times at the cost of extended approximate string matching times. This has been an expected result due to the experiments on self-index based ASM conducted by Russo and colleagues [116]. The comparison of *PTPan* and *BWA* has been repeated based on genome sequence data revealing the same behavior concerning the relevant performance indicators construction time and ASM query times (refer to section 4.3.1).

The main competing approach to *PTPan* is the PT-Server which is part of the ARB software environment [84]. *PTPan* has been developed to replace PT-Server in terms of functionality while dealing with its disadvantages, most important the high main memory demands and the inability to identify insertions or deletions (indels). For different primer/probe based analysis identifying indels is of interest, for example for fluorescence in-situ hybridization (FISH) where indels may lead to false positives recently shown by McIlroy and colleagues [89].

A deep comparison of the PT-Server and *PTPan* has be conducted in [32]. It revealed several common features as well as important differences.

Both deal with ambiguous sequence bases in the source sequence as "N" during construction and as wildcard during string matching (refer to section 2.2.2). The applications like *ProbeDesign* and *ProbeMatch* return meaningful information necessary to evaluate the results (refer to sections 3.3.3 and 3.3.2). Furthermore both indexes are capable of conducting queries in basic- or weighted match mode (refer to section 3.3.2).

Besides the similarities there are important differences. As shown in [32], the approximate string matching capabilities of the PT-Server, utilizing the Hamming-distance metric, are not capable of identifying insertions or deletions (indels). In contrast, due to *PTPan* utilizing the Levenshtein-distance metric, it offers advanced approximate string matching capabilities including indel identification. However, this ability as well as the lower memory requirements achieved by combining the utilization of secondary storage and compression, are at the expense of increased query times compared to the ARB PT-Server. The values presented in [32] revealed a slowdown of factor 4 to 15 for the maximum distance ranging from 0 to 4 mismatches allowed on a 24 GB system and the SILVA SSURef 104 database [113]. Due to optimizations, these values have been lowered for the more recent SILVA SSURef 108 database to a slowdown of 3.6 to 6.5 for one hardware thread and even 1.7 to 3 times for four threads, narrowing the gap significantly.

Finally and most important, *PTPan* has been designed to have low main memory requirements while the memory requirements of the PT-Server prevent its utilization on low memory systems. The test for gene data presented in [32] showed the inability to construct the PT-Server for SILVA SSURef 104 database on a system equipped with 4 GB main memory while *PTPan* can be constructed and utilized efficiently. The evaluation presented in section 4.2.1 approved that the PT-Server faces the same problem for genome sequence databases as well. On a system equipped with 24 GB of main memory, a test database containing genome sequences with more than 4.8 billion bases in total, PT-Server could not be constructed. In contrast *PTPan*, based on a partitioned design in conjunction with stream-compressing the suffix tree on secondary storage, could be constructed in reasonable time of less than six hours. Furthermore the resulting index could be utilized even on a 4 GB system although facing acceptable slowdown of operations.

Additionally in order to keep pace with the fast increase of genome sequence data available, the *PTPan* construction algorithm has been improved significantly by utilizing parallelization. The evaluation results presented in section 4.2.2 reveal that parallel construction offers reasonable speedups on distributed memory systems as well as on shared memory systems. On a system equipped with 24 GB of main memory and 4 processor cores, the construction time for the largest genome test database dropped to less than four hours. The measurements also showed that increasing the number of threads in a shared memory system can lead to a significantly increased amount of partitions to construct with the current partition determination algorithm. As a consequence, the construction time may increase although more threads are utilized. In order to prevent this, an automatic detection of the best partition-to-thread-count-ratio has been implemented. It proofed to be a good estimation choosing the best combination in most cases for gene as well as genome sequence data. Regarding the further increase of the amount of sequence data, the MPI based distributed memory construction algorithm offers the opportunity to construct an index with the help of HPC and employing it on normal desktop systems.

Besides parallelization a *ratio factor* parameter has been integrated which allows the user to provide information about the maximal memory consumption based on the worst case estimation (refer to section 3.4.1). This can lead to less partitions to construct and thus reduce the construction time significantly. The evaluation has shown that the *ratio factor* can be indeed of great use to minimize construction times (refer to section 4.2.3). For example taking a low memory system with 4 GB of main memory, the sequential construction time for the SILVA SSURef 108 database could be lowered by more than factor two from 53 to less than 25 minutes. However, currently the *ratio factor* must be provided by the user as a parameter as it cannot be determined automatically. It is based on prior obtained empirical data. However this is still beneficial if the index needs to be reconstructed, for slowly growing data collections or for homologue sequence databases.

5.3 UMDA Primer/Probe Designer

UMDA Primer/Probe (UPP) Designer, an integrated system for genome sequence data based microbial in-silico diagnostics, has been presented in section 3.5.4. It is based on the *UMDA* framework modules, i.e. the object model, the *database interface (DBI)* and the *search index interface (SII)*. The *DBI* allows flexible access to different heterogeneous sequence data sources, for example utilized by *SII* based index construction. The *SII* further enables the utilization of search index based primer/probe design and evaluation capabilities as they are provided for example by the *PTPan SII* based on the *PTPan* library.

An existing software tool capable of performing analysis on molecular sequence data based on an integrated database is the ARB software environment [84]. Unfortunately ARB suffers from different problems making it unsuitable for further developments extending the base system. The ARB database is a deeply integrated proprietary hierarchical in-memory database design. This implies that at program start an ARB database instance is loaded completely into main memory. Therefore the size of an ARB database is limited by the available hardware. Facing the growth of available genome sequence data outpacing the growth of main memory capacity, this becomes a major drawback. Furthermore ARB has no clear defined object model and database interface. Therefore an easy enhancement or replacement of the database without having to adopt the related software tools is not possible as most of them operate directly on internal data structures of the ARBDB. Finally the ARB database is the only source for the different analysis and primer/probe design and evaluation applications like the ARB PT-Server. It is not possible to access other databases like *EnsEMBL* or *BioSQL* schema based ones (refer to section 2.1.3).

In contrast, the *UPP Designer* avoids these drawbacks according to recommendations of Jagadish and Olken [63]. Leveraging the *UMDA DBI* plugins, *UPP Designer* can access different heterogeneous data sources without modification of the source code. It is now possible to utilize database management systems which do not require to load the complete database into main memory, for example MySQL or PostgreSQL. This enables primer/probe design for large databases even on systems with limited main memory. In addition, by utilizing the *UMDA SII*, it is possible for the *UPP Designer* to employ sophisticated index structures and primer/probe design and evaluation capabilities based on it. For example *PTPan*, which can be constructed and applied with limited main memory as well, can now be utilized with all different data sources accessible by the *UMDA DBI*.

The evaluation of the *UPP Designer* demonstrates the capabilities to design primer and probes for large amounts of data even on systems with limited main memory efficiently (refer to section 4.3.2). For example, based on a *PTPan* index for the cellular prokaryote genomes extracted from the *GenomeReviews release 130*, summing up to a total of more than 4.8 billion bases, *ProbeDesign* for the domain of bacteria could be conducted on a 4 GB computer system in only 74 minutes (refer to section 4.3.2). The same test was conducted with a solid-state drive (SSD) to analyze the impact of this new technology. The results are promising as the time measured was less than 46 minutes, which is a speedup of 1.63 without modifying the software. Further tests on other systems as well as *PTPan SimilaritySearch* based evaluation confirmed this result.

In summary, the evaluation revealed that the *PTPan SII* based mass query applications, accessed from the *UPP Designer* graphical user interface, perform well on different computer systems from commonly equipped notebooks to top tier desktop systems.

6 Summary And Conclusion

The in-silico design of primers and probes based on nucleic acid genome sequence data is of great interest concerning molecular identification and detection of bacteria and viruses which is applied in many areas of research and economy. Unfortunately the currently available software tools for designing primers and probes have reached performance limitations. These are caused by the extremely fast growth of publicly available sequence data of microorganisms which is produced by high throughput sequencing technology.

In this thesis a highly scalable integrated system for microbial in-silico diagnostics based on microbial genome sequence data has been developed.

The first central component is the *Unified Molecular Database Access (UMDA)* framework. It enables the efficient retrieval and management of sequence and annotation data from different heterogeneous data sources with the ability to obtain data sets ranging from small subsets to all available microbial genomes. This is achieved by providing an enhanced object model and an abstract database interface. The abstract database interface has been successfully implemented for different genome database schemas and different database management systems. Furthermore *UMDA* offers unified access to primer/probe design capabilities in form of the abstract search index interface.

To support fast non-heuristic oligonucleotide string matching and applications for primer and probe design based on sequence data, which are required by computational microbial diagnostics, a new memory independent nucleic acid sequence index structure called *PT-Pan* has been developed as second central component. With suffix trees on secondary storage as core structure, it combines partitioning, truncation and a new stream-compression technique to obtain a space efficient index. The evaluation conducted revealed the ability to construct an index for large amounts of genome sequence data efficiently. Furthermore, even if the index outranges the available main memory, it performs well for approximate oligonucleotide string matching, probe design and similarity search functions. *PTPan*, available as standalone library, has been successfully integrated into the ARB software environment. Furthermore *PTPan* has been incorporated into the *UMDA* framework as index underlying a search index interface implementation.

Finally, with the *UMDA Primer/Probe (UPP) Designer* an integrated system for primer and probe design and evaluation based on the *UMDA* framework has been developed. It provides a graphical user interface to utilize the *UMDA SII* application capabilities intuitively. In order to handle the huge amount of available and impending data, the components of this integrated system have been optimized and parallelized to efficiently utilize multi-core-architectures and high performance computers. Evaluation results reveal the success of these efforts.

The *UPP Designer* will allow a large amount of users to continue their work on common desktop computers. In conjunction with the MPI version of *PTPan*, it also enables new HPC-based applications and research not possible so far with the currently available software.

7 Future Work

In order to keep up with the advancements in bioinformatics and the increase of the amount of available nucleic acid genome sequence data, the *UPP Designer* and the components it is based on can be further improved.

As stated by Jagadish and Olken, flat-files are widely utilized as a way of storing nucleic acid sequence data [63]. In addition, the heterogeneity of file formats has recently risen different standardization efforts to ease exchange of data between different applications. With SeqXML [118] and BioXSD [64] new promising formats are available. In order to further facilitate the data access capabilities of the *UMDA* framework, support for these file formats should be integrated. In addition, as collections of flat-files are often utilized as databases according to Jagadish and Olken [63], a *UMDA* database interface implementation based on flat-files may be of interest as well.

In addition, further improvements of the construction algorithm of *PTPan* are of great interest to keep up with the growing amount of available genome sequence data. As the evaluation conducted revealed, constructing a *PTPan* index with more than one thread on a shared memory system does not necessarily result in a reduced construction time due to a higher number of partitions to construct. Although the currently employed partition determination algorithm offers a good estimation of the best thread count and partition distribution to use, a sophisticated partition determination algorithm for shared memory systems is of interest in order to further reduce the overall construction times. One possibility would be to develop an algorithm capable of finding an ideal distribution of partitions taking into account the amount of main memory available as well as the thread count. This problem is not trivial as it is a knapsack problem and therefore NP-complete to solve exactly.

In addition, it is of interest to research the possibilities of an automatic optimization of the worst case memory requirements estimation based on the source sequence data. Currently there is only the possibility to provide an empirical obtained *ratio factor* parameter to manually correct the estimation.

Optimizing and enhancing the primer/probe applications offered by *PTPan* and the integrated system is of interest as well. Users could benefit by extended application capabilities and reduced runtime.

Furthermore, the length of probes which can be designed by *PTPan* is limited to the longest truncation depth of the suffix tree, currently 27 at maximum. Although this is sufficient for many applications such as polymerase chain reaction (PCR) or fluorescence in-situ hybridization (FISH) [3], longer probes would increase the field of application. For example it would facilitate the development of microarrays used for gene expression analysis. The optimal length in terms of sensitivity and specificity for oligonucleotide probes in this field is between 50 and 60 bases [82]. Extending the maximum suffix tree depth to for example 100 would make designing longer probes possible.

Furthermore, with increasing source sequence data amounts, the index size will continue to grow as well. In order to speed up oligonucleotide string matching and the mass query based applications, parallel computations on the index are of interest. One possibility is to employ distributed computing to speed up pattern matching for a large number of patterns on cluster computers as demonstrated by Bader and colleagues for the ARB PT-Server utilizing DUP, a framework for parallel stream processing [5]. This could be the basic building block of a server component for primer/probe design. Another option is to further research the parallel query capabilities on shared memory systems in conjunction with solid state drives.

Finally the *UMDA Primer/Probe (UPP) Designer* graphical user interface offers space for improvements as well. It could function as the client application for accessing the above proposed server component for primer and probe design. Integrating the MPI version of *PTPan* as alternative index construction option would simplify the construction of an search index for large amounts of genome sequence data from within the *UPP Designer* by reducing the number of manual steps a user has to conduct. In addition, the integration of a phylogenetic tree viewer into the *UPP Designer* primer/probe design process would enable an easier way to select sequence entries related to organisms within certain domains of life.

List of Figures

List of Tables

List of Algorithms

Abbreviations

API	Application Programming Interface
ASM	Approximate String Matching
BWT	Burrows-Wheeler Transformation
CAS	Compare-And-Swap
CMD	Computational Molecular Diagnostics
CSA	Compressed Suffix Array
CST	Compressed Suffix Tree
DBI	DataBase Interface
DBMS	DataBase Management System
DFS	Depth First Search
DLP	Data Level Parallelism
DNA	DeoxyriboNucleic Acid
DNA4	DNA alphabet without ambigiuos character N
DNA5	DNA alphabet including ambigiuos character N
HPC	High Performance Computing
ILP	Instruction Level Parallelism
IUPAC	International Union of Pure and Applied Chemistry
LZ	Lempel-Ziv
MIMD	Multiple Instructions, Multiple Data
MPI	Message Passing Interface
MPMD	Multiple Program, Multiple Data
OBF	Open Bioinformatics Foundation
OpenMP	Open Multi Processing
RDBMS	Relational DataBase Management System
REST	REpresentational State Transfer
RNA	RiboNucleic Acid
SA	Suffix Array
SDK	Software Development Kit
SII	Search Index Interface
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processor
SOAP	Simple Object Access Protocol
SPMD	Single Program, Multiple Data
SSD	Solid State Drive
ST	Suffix Tree
TLP	Task Level Parallelism
UMDA	Unified Molecular Data Access
UPP Designer	UMDA Primer/Probe Designer
WASM	Weighted Approximate String Matching

XML	Extensible Markup Language

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2 , Issue 1:53–86, March 2004.

[2] Shameem Akhter and Jason Roberts. Avoiding Classic Threading Problems. http://www.drdobbs.com, June 2011.

[3] Rudolf Amann and Bernhard M. Fuchs. Single-cell identification in microbial communities by improved fluorescence in situ hybridization techniques. *Nature Reviews Microbiology*, 6:339–348, May 2008.

[4] Linda A. Amaral-Zettler, Elizabeth A. McCliment, Hugh W. Ducklow, and Susan M. Huse. A method for studying protistan diversity using massively parallel sequencing of V9 hypervariable regions of small-subunit ribosomal RNA genes. *PLoS ONE*, 4:e6372, June 2009.

[5] Kai Bader, Tilo Eißler, Nathan Evans, Chris GauthierDickey, Christian Grothoff, Krista Grothoff, Jeff Keene, Harald Meier, Craig Ritzdorf, and Matthew Rutherford. Distributed Stream Processing with DUP. In *Network and Parallel Computing*, volume 6289 of *Lecture Notes in Computer Science*, pages 232–246. Springer Berlin / Heidelberg, 2010.

[6] Kai Christian Bader, Christian Grothoff, and Harald Meier. Comprehensive and relaxed search for oligonucleotide signatures in hierarchically clustered sequence datasets. *Bioinformatics*, 27(11):1546–1554, 2011.

[7] A. Barbosa-Silva, E. Pafilis, J.M. Ortega, and R. Schneider. Development of SRS.php, a Simple Object Access Protocol-based library for data acquisition from integrated biological databases. *Genetics and Molecular Research*, 6(4):1142–1150, December 2007.

[8] Marina Barsky, Ulrike Stege, and Alex Thomo. A survey of practical algorithms for suffix tree construction in external memory. *Software: Practice and Experience*, 40(11):965–988, 2010.

[9] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers. GenBank. *Nucleic Acids Research*, 39(suppl 1):D32–D37, 2011.

[10] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. GenBank. *Nucleic Acids Research*, 36(Database issue):D25–D30, 2008.

[11] Documentation of BioJava. `http://biojava.org/`, last access at 14. December 2011.

[12] Documentation of BioPerl. `http://www.bioperl.org/`, last access at 14. December 2011.

[13] Documentation of BioPython. `http://www.biopython.org`, last access at 14. December 2011.

[14] Documentation of BioSQL. `http://www.biosql.org/`, last access at 22. December 2011.

[15] BioWisdom Ltd. `http://www.biowisdom.com/`, last access at 26. December 2011.

[16] Daniel Blankenberg, Nathan Coraor, Gregory Von Kuster, James Taylor, and Anton Nekrutenko. Integrating diverse databases into an unified analysis framework: a Galaxy approach. *Database*, 2011:online, 2011.

[17] Jörg Böhnel. Parallelisierung der Suche nach diagnostischen Signaturen in großen molekularen Sequenzdatenbanken auf verteilten Computersystemen. Master's thesis, Technische Universität München, Fakultät für Informatik, July 2009.

[18] Boost C++ libraries. `http://www.boost.org/`, last access at 14. December 2011.

[19] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Systems Research Center (SRC), 1994.

[20] Ermai Cai. A Client-Application for Web based Searches in Molecular Sequence Databases. Master's thesis, Technische Universität München, Fakultät für Informatik, January 2011.

[21] Chado. `http://gmod.org/wiki/Chado`, last access at 14. December 2011.

[22] Chunxi Chen and Bertil Schmidt. Constructing large suffix trees on a computational grid. *Journal of Parallel and Distributed Computing*, 66(12):1512–1523, 2006. Special Issue: Grids in Bioinformatics and Computational Biology.

[23] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

[24] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[25] J. R. Cole, Q. Wang, E. Cardenas, J. Fish, B. Chai, R. J. Farris, A. S. Kulam-Syed-Mohideen, D. M. McGarrell, T. Marsh, G. M. Garrity, and J. M. Tiedje. The Ribosomal Database Project: improved alignments and new tools for rRNA analysis. *Nucleic Acids Research*, 37(suppl 1):D141–D145, 2009.

[26] NVIDIA CUDA. `http://www.nvidia.com/cuda`, last access at 03. January 2012.

[27] EBI Dbfetch. `http://www.ebi.ac.uk/Tools/dbfetch/`, last access at 05. September 2011.

[28] Erika De Francesco, Giuliana Di Santo, Luigi Palopoli, and Simona Rombo. A Summary of Genomic Databases: Overview and Discussion. In Amandeep Sidhu and Tharam Dillon, editors, *Biomedical Data and Applications*, volume 224 of *Studies in Computational Intelligence*, pages 37–54. Springer Berlin / Heidelberg, 2009.

[29] T. Z. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. L. Brodie, K. Keller, T. Huber, D. Dalevi, P. Hu, and G. L. Andersen. Greengenes, a Chimera-Checked 16S rRNA Gene Database and Workbench Compatible with ARB. *Applied and Environmental Microbiology*, 72(7):5069–5072, 2006.

[30] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9: 11:1–9, January 2008.

[31] Julien Dutheil, Sylvain Gaillard, Eric Bazin, Sylvain Glemin, Vincent Ranwez, Nicolas Galtier, and Khalid Belkhir. Bio++: a set of C++ libraries for sequence analysis, phylogenetics, molecular evolution and population genetics. *BMC Bioinformatics*, 7(1):188, 2006.

[32] Tilo Eißler, Christopher P. Hodges, and Harald Meier. PTPan – overcoming memory limitations in oligonucleotide string matching for primer/probe design. *Bioinformatics*, 27(20):2797–2805, August 2011.

[33] EMBL file formats. `http://www.ebi.ac.uk/ena/about/embl_bank_format`, last access at 22. December 2011.

[34] EnsEMBL. `http://www.ensembl.org/`, last access at 14. December 2011.

[35] EnsEMBL MySQL server. `http://www.ensembl.org/info/data/mysql.html`, last access at 31. December 2011.

[36] T. Etzold, A. Ulyanov, and P. Argos. SRS: information retrieval system for molecular biology data banks. *Methods Enzymol*, 266:114–128, 1996.

[37] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society.

[38] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21:948–960, September 1972.

[39] Michael Y. Galperin and Guy R. Cochrane. The 2011 Nucleic Acids Research Database Issue and the online Molecular Biology Database Collection. *Nucleic Acids Research*, 39(suppl 1):D1–D6, 2011.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[41] Genome Reviews. `http://www.ebi.ac.uk/GenomeReviews/`, last access at 28. January 2012.

[42] Amol Ghoting and Konstantin Makarychev. Serial and parallel methods for i/o efficient suffix tree construction. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 827–840, New York, NY, USA, 2009. ACM.

[43] Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

[44] Andreas Gogol-Döring. *SeqAn - a generic software library for sequence analysis*. PhD thesis, FB Mathematik und Informatik - FU Berlin, 2009.

[45] GOLD (Genomes OnLine Database). `http://www.genomesonline.org/`, last access at 07. February 2012.

[46] Naohisa Goto, Pjotr Prins, Mitsuteru Nakao, Raoul Bonnal, Jan Aerts, and Toshiaki Katayama. BioRuby: Bioinformatics software for the Ruby programming language. *Bioinformatics*, btq:475, August 2010.

[47] Manolo Gouy and Stéphane Delmotte. Remote access to ACNUC nucleotide and protein sequence databases at PBIL. *Biochimie*, 90(4):555 – 562, April 2008.

[48] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A High-Performance, Heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[49] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22:789–828, September 1996.

[50] Roberto Grossi. A quick tour on suffix arrays and compressed suffix arrays. *Theoretical Computer Science*, 412(27):2964–2973, June 2011.

[51] Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[52] Jonathan M. Guberman, J. Ai, O. Arnaiz, Joachim Baran, Andrew Blake, Richard Baldock, Claude Chelala, David Croft, Anthony Cros, Rosalind J. Cutts, A. Di Génova, Simon Forbes, T. Fujisawa, E. Gadaleta, D. M. Goodstein, Gunes Gundem, Bernard Haggarty, Syed Haider, Matthew Hall, Todd Harris, Robin Haw, S. Hu, Simon Hubbard, Jack Hsu, Vivek Iyer, Philip Jones, Toshiaki Katayama, R. Kinsella, Lei Kong, Daniel Lawson, Yong Liang, Nuria Lopez-Bigas, J. Luo, Michael Lush, Jeremy Mason, Francois Moreews, Nelson Ndegwa, Darren Oakley, Christian Perez-Llamas, Michael Primig, Elena Rivkin, S. Rosanoff, Rebecca Shepherd, Reinhard Simon, B. Skarnes, Damian Smedley, Linda Sperling, William Spooner, Peter Stevenson, Kevin Stone, J. Teague, Jun Wang, Jianxin Wang, Brett Whitty, D. T. Wong, Marie Wong-Erasmus, L. Yao, Ken Youens-Clark, Christina Yung, Junjun Zhang, and Arek

Kasprzyk. BioMart Central Portal: an open database network for the biological community. *Database*, 2011:bar041, August 2011.

[53] Richard W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29:147–160, 1950.

[54] Waqar Haque, Alex Aravind, and Bharath Reddy. Pairwise sequence alignment algorithms: a survey. In *Proceedings of the 2009 conference on Information Science, Technology and Applications*, ISTA '09, pages 96–103, New York, NY, USA, 2009. ACM.

[55] Christopher P. Hodges. Distributed data structures for efficient molecular sequence analysis. Master's thesis, Technische Universität München, Fakultät für Informatik, November 2003.

[56] R.C.G. Holland, T. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Dräger, A. Yates, M. Heuer, and M.J. Schreiber. BioJava: an Open-Source Framework for Bioinformatics. *Bioinformatics*, 24(18):2096–2097, 2008.

[57] T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L. Clark, T. Cox, J. Cuff, V. Curwen, T. Down, R. Durbin, E. Eyras, J. Gilbert, M. Hammond, L. Huminiecki, A. Kasprzyk, H. Lehvaslaiho, P. Lijnzaad, C. Melsopp, E. Mongin, R. Pettett, M. Pocock, S. Potter, A. Rust, E. Schmidt, S. Searle, G. Slater, J. Smith, W. Spooner, A. Stabenau, J. Stalker, E. Stupka, A. Ureta-Vidal, I. Vastrik, and M. Clamp. The Ensembl genome database project. *Nucleic Acids Research*, 30(1):38–41, 2002.

[58] David Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 11:91–99, September 1952.

[59] Portable Hardware Locality (hwloc). `http://www.open-mpi.org/projects/hwloc/`, last access at 09. January 2011.

[60] International Nucleotide Sequence Database Collaboration . `http://www.insdc.org/`, last access at 22. December 2011.

[61] Intel Threading Building Blocks. `http://threadingbuildingblocks.org/`, last access at 15. December 2011.

[62] International Union of Pure and Applied Chemistry. Abbreviations and symbols for nucleic acids, polynucleotides and their constituents (Rules approved 1974). Technical Report 40 (3), 277–331, Pure and Applied Chemistry, 1974.

[63] H. V. Jagadish and Frank Olken. Database management for life sciences research. *ACM SIGMOD Record*, 33(2):15–20, 2004.

[64] Matus Kalas, Pal Puntervoll, Alexandre Joseph, Edita Bartaseviciute, Armin Töpfer, Prabakar Venkataraman, Steve Pettifer, Jan Christian Bryne, Jon Ison, Christophe Blanchet, Kristoffer Rapacki, and Inge Jonassen. BioXSD: the common data-exchange format for everyday bioinformatics web services. *Bioinformatics*, 26(18):i540–i546, 2010.

[65] Eli Kaminuma, Takehide Kosuge, Yuichi Kodama, Hideo Aono, Jun Mashima, Takashi Gojobori, Hideaki Sugawara, Osamu Ogasawara, Toshihisa Takagi, Kousaku Okubo, and Yasukazu Nakamura. DDBJ progress report. *Nucleic Acids Research*, 39(suppl 1):D22–D27, January 2011.

[66] P. J. Kersey, D. Lawson, E. Birney, P. S. Derwent, M. Haimel, J. Herrero, S. Keenan, A. Kerhornou, G. Koscielny, A. Kähäri, R. J. Kinsella, E. Kulesha, U. Maheswari, K. Megy, M. Nuhn, G. Proctor, D. Staines, F. Valentin, A. J. Vilella, and A. Yates. Ensembl genomes: Extending ensembl across the taxonomic space. *Nucleic Acids Research*, 38(Database Issue):D563–D569, January 2010.

[67] Paul Kersey, Lawrence Bower, Lorna Morris, Alan Horne, Robert Petryszak, Carola Kanz, Alexander Kanapin, Ujjwal Das, Karine Michoud, Isabelle Phan, Alexandre Gattiker, Tamara Kulikova, Nadeem Faruque, Karyn Duggan, Peter Mclaren, Britt Reimholz, Laurent Duret, Simon Penel, Ingmar Reuter, and Rolf Apweiler. Integr8 and Genome Reviews: integrated views of complete genomes and proteomes. *Nucleic Acids Research*, 33(Database Issue):D297–D302, 2005.

[68] Jeong Myeong Kim, Hyo Jung Lee, Sun Young Kim, Jae Jun Song, Woojun Park, and Che Ok Jeon. Analysis of the fine-scale population structure of "Candidatus accumulibacter phosphatis" in enhanced biological phosphorus removal sludge, using fluorescence in situ hybridization and flow cytometric sorting. *Applied and Environmental Microbiology*, 76:3825–3835, June 2010.

[69] Rhoda J. Kinsella, Andreas Kähäri, Syed Haider, Jorge Zamora, Glenn Proctor, Giulietta Spudich, Jeff Almeida-King, Daniel Staines, Paul Derwent, Arnaud Kerhornou, Paul Kersey, and Paul Flicek. Ensembl BioMarts: a hub for data retrieval across taxonomic space. *Database*, 2011:online, July 2011.

[70] Tamara Kulikova, Philippe Aldebert, Nicola Althorpe, Wendy Baker, Kirsty Bates, Paul Browne, Alexandra van den Broek, Guy Cochrane, Karyn Duggan, Ruth Eberhardt, Nadeem Faruque, Maria Garcia-Pastor, Nicola Harte, Carola Kanz, Rasko Leinonen, Quan Lin, Vincent Lombard, Rodrigo Lopez, Renato Mancuso, Michelle McHale, Francesco Nardone, Ville Silventoinen, Peter Stoehr, Guenter Stoesser, Mary Ann Tuli, Katerina Tzouvara, Robert Vaughan, Dan Wu, Weimin Zhu, and Rolf Apweiler. The EMBL Nucleotide Sequence Database . *Nucleic Acids Research*, 32(Database issue):D27–D30, 2004.

[71] Fabian Kulla and Peter Sanders. Scalable Parallel Suffix Array Construction. *Lecture Notes in Computer Science*, 4192:22–29, 2006.

[72] Yadhu Kumar, Ralf Westram, Sebastian Behrens, Bernhard Fuchs, Frank Oliver Glöckner, Rudolf Amann, Harald Meier, and Wolfgang Ludwig. Graphical representation of ribosomal RNA probe accessibility data using ARB software package. *BMC Bioinformatics*, 6:61, March 2005.

[73] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12, 2004.

[74] Hsiao Ping P. Lee, Tzu-Fang F. Sheu, and Chuan Yi Y. Tang. A parallel and incremental algorithm for efficient unique signature discovery on DNA databases. *BMC bioinformatics*, 11:132, 2010.

[75] Inbok Lee, Costas Iliopoulos, and Syng-Yup Ohn. Transformation of Suffix Arrays into Suffix Trees on the MPI Environment. In Aijun An, Jerzy Stefanowski, Sheela Ramanna, Cory Butz, Witold Pedrycz, and Guoyin Wang, editors, *Rough Sets, Fuzzy Sets, Data Mining and Granular Computing*, volume 4482 of *Lecture Notes in Computer Science*, pages 248–255. Springer Berlin / Heidelberg, 2007.

[76] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[77] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[78] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[79] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[80] libpqxx. `http://pqxx.org/development/libpqxx/`, last access at 15. December 2011.

[81] Konstantinos Liolios, I-Min A. Chen, Konstantinos Mavromatis, Nektarios Tavernarakis, Philip Hugenholtz, Victor M. Markowitz, and Nikos C. Kyrpides. The Genomes On Line Database (GOLD) in 2009: status of genomic and metagenomic projects and their associated metadata. *Nucleic Acids Research*, 38(suppl 1):D346–D354, 2009.

[82] Hongfang Liu, Ionut Bebu, and Xin Li. Microarray probes and probe sets. *Frontiers in bioscience elite edition*, 2:325–338, 2010.

[83] Alexander Loy, Roland Arnold, Patrick Tischler, Thomas Rattei, Michael Wagner, and Matthias Horn. probeCheck - a central resource for evaluating oligonucleotide probe coverage and specificity. *Environmental Microbiology*, 10(10):2894–2898., October 2008.

[84] Wolfgang Ludwig, Oliver Strunk, Ralf Westram, Lothar Richter, Harald Meier, Yadhukumar, Arno Buchner, Tina Lai, Susanne Steppi, Gangolf Jobb, Wolfram Förster, Igor Brettske, Stefan Gerber, Anton W. Ginhart, Oliver Gross, Silke Grumann, Stefan Hermann, Ralf Jost, Andreas König, Thomas Liss, Ralph Lüßmann, Michael May, Björn Nonhoff, Boris Reichel, Robert Strehlow, Alexandros Stamatakis, Norbert Stuckmann, Alexander Vilbig, Michael Lenke, Thomas Ludwig, Arndt Bode, and Karl-Heinz Schleifer. ARB: a software environment for sequence data. *Nucleic Acids Research*, 32(4):1363–1371, February 2004.

[85] Daniel MacLean, Jonathan D. G. Jones, and David J. Studholme. Application of 'next-generation' sequencing technologies to microbial genetics. *Nature Reviews Microbiology*, 7:287–296, April 2009.

[86] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[87] Harry Mangalam. The Bio* toolkits – a brief overview. *Briefings in Bioinformatics*, 3(3):296–302, 2002.

[88] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. ERA: efficient serial and parallel suffix tree construction for very long strings. *Proc. VLDB Endow.*, 5:49–60, September 2011.

[89] Simon J. McIlroy, Daniel Tillett, Steve Petrovski, and Robert J. Seviour. Non-target sites with single nucleotide insertions or deletions are frequently found in 16S rRNA sequences and can lead to false positives in fluorescence in situ hybridization (FISH). *Environmental Microbiology*, 13(1):33 – 47, January 2011.

[90] Michael L. Metzker. Sequencing technologies - the next generation. *Nature Review Genetics*, 11(1):31–46, January 2010.

[91] P. D. Michailidis and K. G. Margaritis. On-line string matching algorithms: survey and experimental results. *International Journal of Computer Mathematics*, 76(4):411–434, 2001.

[92] Alistair Moffat, Simon J. Puglisi, and Ranjan Sinha. Reducing Space Requirements for Disk Resident Suffix Arrays . In *Database Systems for Advanced Applications*, volume 5463/2009 of *Lecture Notes in Computer Science*, pages 730–744. Springer Berlin / Heidelberg, March 2009.

[93] Donald R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15:514–534, October 1968.

[94] MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2/`, last access at 15. December 2011.

[95] Message Passing Interface Forum. `http://www.mpi-forum.org`, last access at 15. December 2011.

[96] Christopher J. Mungall, David B. Emmert, , and The FlyBase Consortium. A Chado case study: an ontology-based modular schema for representing genome-associated biological information. *Bioinformatics*, 23(13):i337–i346, 2007.

[97] MySQL. `http://www.mysql.com/`, last access at 14. December 2011.

[98] MySQL++. `http://tangentsoft.net/mysql++/`, last access at 15. December 2011.

[99] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, March 2001.

[100] Gonzalo Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms*, 2:87–114, March 2004.

[101] Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen, and Jorma Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24:19–27, 2001.

[102] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39:–, April 2007.

[103] NCBI C++ Toolkit. `http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/`, last access at 31. December 2011.

[104] NCBI Entrez Programming Utilities Help. `http://www.ncbi.nlm.nih.gov/books/NBK25501/`, last access at 31. December 2011.

[105] Open Biological Database Access (OBDA). `http://obda.open-bio.org/`, last access at 26. December 2011.

[106] The Open Bioinformatics Foundation. `http://www.open-bio.org/`, last access at 25. August 2011.

[107] OpenCL. `http://www.khronos.org/opencl/`, last access at 03. January 2011.

[108] OpenMP. `http://www.openmp.org`, last access at 15. December 2011.

[109] SRS URL API (Perl). `http://sourceforge.net/apps/trac/srsurlapi/`, last access at 26. December 2011.

[110] Adam M. Phillippy, Kunmi Ayanbule, Nathan J. Edwards, and Steven L. Salzberg. Insignia: a DNA signature search web server for diagnostic assay development. *Nucl. Acids Res.*, 37(suppl_2):W229–234, 2009.

[111] Adam M Phillippy, Jacquline A Mason, Kunmi Ayanbule, Daniel D Sommer, Elisa Taviani, Anwar Huq, Rita R Colwell, Ivor T Knight, and Steven L Salzberg. Comprehensive DNA Signature Discovery and Validation. *PLoS Comput Biol.*, 3(5):e98, May 2007.

[112] PostgreSQL. `http://www.postgresql.org/`, last access at 28. December 2011.

[113] Elmar Pruesse, Christian Quast, Katrin Knittel, Bernhard M. Fuchs, Wolfgang Ludwig, Jörg Peplies, and Frank Oliver Glöckner. SILVA: a comprehensive online resource for quality checked and aligned ribosomal RNA sequence data compatible with ARB. *Nucleic Acids Research*, 35(21):7188–7196, 2007.

[114] Kim D. Pruitt, Tatiana Tatusova, William Klimke, and Donna R. Maglott. NCBI Reference Sequences: current status, policy and new initiatives. *Nucleic Acids Research*, 37(suppl 1):D32–D36, 2009.

[115] The Qt Project. `http://qt-project.org/`, last access at 09. January 2012.

[116] Luís M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales. Approximate String Matching with Compressed Indexes. *Algorithms*, 2(3):1105–1136, 2009.

[117] Wolfram Saenger. *Principles of Nucleic Acid Structure*. Springer-Verlag, 1984.

[118] Thomas Schmitt, David N. Messina, Fabian Schreiber, and Erik L.L. Sonnhammer. SeqXML and OrthoXML: standards for sequence and orthology information. *Briefings in Bioinformatics*, 2011:online, 2011.

[119] Susan Schönmann, Alexander Loy, Céline Wimmersberger, Jens Sobek, Catharine Aquino, Peter Vandamme, Beat Frey, Hubert Rehrauer, and Leo Eberl. 16S rRNA gene-based phylogenetic microarray for simultaneous identification of members of the genus Burkholderia. *Environmental Microbiology*, 11:779–800, April 2009.

[120] Marcel H. Schulz, Sebastian Bauer, and Peter N. Robinson. The generalised k-Truncated Suffix Tree for time-and space-efficient searches in multiple DNA or protein sequences. *International Journal of Bioinformatics Research and Applications*, 4(1):81–95, 2008.

[121] Sohrab P Shah, Yong Huang, Tao Xu, Macaire MS Yuen, John Ling, and BF Francis Ouellette. Atlas – a data warehouse for integrative bioinformatics. *BMC Bioinformatics*, 6:34, February 2005.

[122] Damian Smedley, Syed Haider, Benoit Ballester, Richard Holland, Darin London, Gudmundur Thorisson, and Arek Kasprzyk. BioMart - biological queries made easy. *BMC Genomics*, 10(1):22, 2009.

[123] SQLite. `http://www.sqlite.org/`, last access at 28. December 2011.

[124] SSAHA2 website. `http://www.sanger.ac.uk/resources/software/ssaha2.html`, last access at 04. January 2011.

[125] A Stabenau, G. McVicker, C. Melsopp, G. Proctor, M. Clamp, and E. Birney. The Ensembl core software libraries. *Genome Research*, 14(5):929–933, May 2004.

[126] Jason E. Stajich, David Block, Kris Boulez, Steven E. Brenner, Stephen A. Chervitz, Chris Dagdigian, Georg Fuellen, James G.R. Gilbert, Ian Korf, Hilmar Lapp, Heikki Lehväslaiho, Chad Matsalla, Chris J. Mungall, Brian I. Osborne, Matthew R. Pocock, Peter Schattner, Martin Senger, Lincoln D. Stein, Elia Stupka, Mark D. Wilkinson, and Ewan Birney. The Bioperl Toolkit: Perl Modules for the Life Sciences. *Genome Research*, 12:1611–1618, 2002.

[127] Jason E. Stajich and Hilmar Lapp. Open source tools and toolkits for bioinformatics: significance, and where are we? *Briefings in Bioinformatics*, 7(3):287–296, August 2006.

[128] Lincoln Stein. Genome annotation: from sequence to biology. *Nature Reviews Genetics*, 2:493–503, July 2001.

[129] P. Sterk, P.J. Kersey, and R. Apweiler. Genome Reviews: Standardizing Content and Representation of Information about Complete Genomes. *OMICS*, 10(2):114–118, 2006.

[130] Oliver Strunk. *ARB: Development of a computer programm to collect, administer and analyse nucleic and amino acid sequences*. PhD thesis, Technische Universität München, Fakultät für Chemie, 2001.

[131] Hideaki Sugawara, Takashi Abe, Takashi Gojobori, and Yoshio Tateno. DDBJ working on evaluation and classification of bacterial genes in INSDC. *Nucleic Acids Research*, 35(Database issue):D13–D15, January 2007.

[132] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[133] Thoralf Töpel, Benjamin Kormeier, Andreas Klassen, and Ralf Hofestädt. BioDWH: A Data Warehouse Kit for Life Science Data Integration. *Journal of Integrative Bioinformatics*, 5(2):93, August 2008.

[134] Dimitris Tsirogiannis and Nick Koudas. Suffix tree construction algorithms on modern hardware. *ACM International Conference Proceeding Series*, 426(Proceedings of the 13th International Conference on Extending Database Technology):263–274, March 2010.

[135] Esko Ukkonen. Approximate string matching over suffix trees. In *Proceedings of the 4th annual symposiumon combinatorial pattern matching, number 684 in lecture notes in computer science*, pages 228–242. Springer-Verlag, 1993.

[136] Denis Vakatov, editor. *The NCBI C++ Toolkit Book*, volume June 2010. National Center for Biotechnology Information (US), 2010.

[137] R. Bruce Wallace, J. Shaffer, R.F. Murphy, J. Bonner, T. Hirose, and K. Itakura. Hybridization of synthetic oligodeoxyribonucleotides to ΦX 174 DNA: the effect of single base pair mismatch. *Nucleic Acids Research*, 6(11):3543–3558, 1979.

[138] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

[139] EBI WSDbfetch (SOAP). `http://www.ebi.ac.uk/Tools/webservices/services/dbfetch`, last access at 31. December 2011.

[140] L. Safak Yilmaz, Lindsey I. Bergsven, and Daniel R. Noguera. Systematic evaluation of single mismatch stability predictors for fluorescence in situ hybridization. *Environmental Microbiology*, 10(10):2872–2885, October 2008.

[141] Bing Zhang and Zhenglin Huang. A new parallel suffix tree construction algorithm . In *Proceedings of the 2011 3rd IEEE International Conference on Communication Software and Networks (ICCSN)*, ICCSN '11, pages 143–147, September 2011.

[142] Di Zhang, Yunquan Zhang, Shengfei Liu, and Xiaodi Huang. Parallelization of FM-Index. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pages 169–173, Washington, DC, USA, 2008. IEEE Computer Society.

[143] Junjun Zhang, Syed Haider, Joachim Baran, Anthony Cros, Jonathan M. Guberman, Jack Hsu, Yong Liang, Long Yao, and Arek Kasprzyk. BioMart: a data federation framework for large collaborative projects. *Database*, 2011:bar038, 2011.

[144] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.