TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur

Algorithms and Computer Architectures for Evolutionary Bioinformatics

Nikolaos Ch. Alachiotis

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. B. Rost
Prüfer der Dissertation:	1. Univ.-Prof. Dr. A. Stamatakis (Karlsruher Institut für Technologie)
	2. Univ.-Prof. Dr. A. Bode
	3. Univ.-Prof. Dr. A. Dollas (TU Kreta/Griechenland)

Die Dissertation wurde am 27.06.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.10.2012 angenommen.

Algorithms and Computer Architectures for Evolutionary Bioinformatics

Nikolaos Ch. Alachiotis

Abstract

Many algorithms in the field of evolutionary Bioinformatics have excessive computational requirements. This holds not only because of the continuous accumulation of molecular sequence data, which is driven by significant advances in wet-lab sequencing technologies, but it is also due to the high computational demands of the employed kernels.

This dissertation presents new reconfigurable architectures to accelerate parsimony- and likelihood-based phylogenetic tree reconstruction, as well as effective techniques to speed up the execution of a phylogeny-aware alignment kernel on general-purpose graphics processing units. It also introduces a novel technique to conduct efficient phylogenetic tree searches on alignments with missing data. In addition, a highly optimized software implementation for the ω statistic, which is used to detect complete selective sweeps in population genetic data using linkage-disequilibrium patterns of single nucleotide polymorphisms in multiple sequence alignments, is described and made available.

Acknowledgments

Many people have contributed to the success of this work. I am particularly grateful to Dr. Alexandros Stamatakis who has been supporting me since my undergraduate studies, and spent a great effort on supervising my work. I would also like to thank Prof. Dr. Arndt Bode for co-supervising my thesis and Prof. Apostolos Dollas for being part of my PhD committee. I am extremely grateful to all my colleagues in the Exelixis lab (Simon A. Berger, Pavlos Pavlidis, Fernando Izquierdo, Andre J. Aberer, Jiajie Zhang, Solon Pissis, Kassian Kobert, and Tomas Flouris), who offered advice and support when it was needed. Finally, I want to thank my parents and my sister who have been particularly supportive during my time in Germany.

My position was funded under the auspices of the Emmy-Noether program by the German Science Foundation (DFG, STA 860/2).

Contents

Chapter 1

Introduction

This introductory chapter provides the motivation for conducting research to improve performance of evolutionary Bioinformatics kernels, summarizes the scientific contribution of this work, and describes the structure of the thesis.

1.1 Motivation

Evolution can be seen as the change in the genetic composition of a biological population over time. As a result of the evolutionary process, successive generations appear to have different phenotypic traits such as behavior, morphology, and development. Charles Robert Darwin (1809-1882) was the first to propose a scientific theory of evolution. Darwin established that life on earth originated and evolved from a universal common ancestor 3.7 billion years ago, and explained evolution as the result of a process that he called natural selection. Natural selection takes place in an environment when members of a population that die are replaced by offsprings that are better adapted to survive and reproduce.

The starting point to conduct a genetic analysis of evolution is the genetic material (DNA) of the organisms under investigation. Initially, DNA sequences are extracted from the organisms using sequencing technology. Thereafter, these sequences are aligned using multiple sequence alignment tools and the output of such tools, a multiple sequence alignment (MSA), is then used for further downstream analyses. Phylogenetic studies can be used to determine how a virus spreads over the globe [134] or to describe major shifts in the diversification rates of plants [171]. Population genetics can be used to infer demographic information such as expansion, migration, mutation, and recombination rates in a population, or the location and intensity of selection processes within a genome.

Recent developments in sequencing technologies, such as Next-Generation Sequencing (NGS), have led to an accelerating accumulation of molecular sequence data, since entire genomes can be sequenced quickly, accurately, and more importantly, in a cost-effective way. Several evolutionary Bioinformatics kernels are known to be particularly expensive in terms of computations required. A representative example is the phylogenetic likelihood function (PLF),

which heavily relies on floating-point arithmetics and is used to calculate the likelihood score of phylogenetic tree topologies.

Evidently, Bioinformatics software tools that rely on compute-intensive kernels exhibit long runtimes and usually require excessive computational resources or memory space to execute. Despite the fact that the end of Moore's law has been proclaimed several times in the past decades, the number of transistors that can be placed inexpensively on an integrated circuit still doubles approximately every two years. The current pace of molecular data accumulation is approximately analogous to what Moore's law predicts for modern microprocessor architectures, thus leading to a very slowly decreasing gap—the *bio-gap*—between the increase of available sequence data and microprocessor performance (see Figure 1.1).

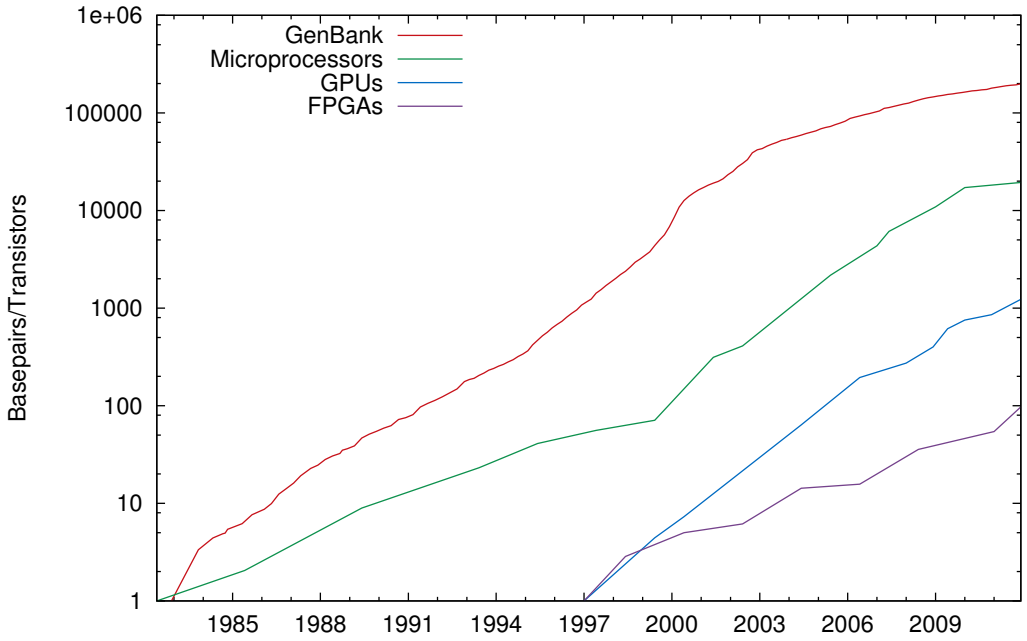


Figure 1.1: Growth of molecular data available in GenBank as well as of number of transistors in microprocessors, GPUs, and FPGAs according to Moore's Law.

Fortunately, the computing landscape is experiencing an evolution of its own with the emergence of more powerful processing elements such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and multi-core CPUs. Parallelism, combined with multi-core technology, represents the primary approach to increase CPU performance nowadays. As can be observed in the latest Top500 list, High-Performance Computing (HPC) systems are steadily transforming from "classic" supercomputers to large clusters of multi-core processors. Additionally, GPUs and FPGAs are gradually being established as alternatives, since they can serve as co-processors to offload computationally intensive kernels. Certain algorithms witness significant performance improvements (more than one order of magnitude) when ported and

executed on such architectures.

Clearly, the immense accumulation of molecular data that needs to be processed and the increased computational demands of the employed kernels in evolutionary Bioinformatics generate a challenge: How to effectively exploit new emerging technologies and/or devise novel algorithmic solutions to accelerate computational kernels, reduce memory requirements, and eventually boost the capacity of modern software tools to keep up with the molecular data growth?

1.2 Scientific contribution

This dissertation comprises scientific contributions to the disciplines of phylogenetics and population genetics by introducing novel algorithms and presenting dedicated computer architectures to accelerate widely used kernels and/or reduce their memory requirements.

In computational phylogenetics, commonly used approaches to reconstruct evolutionary trees are maximum parsimony, maximum likelihood, and Bayesian techniques. Maximum parsimony strives to find the phylogenetic tree that explains the evolutionary history of organisms by the least number of mutations. A reconfigurable co-processor to accelerate the parsimony kernel is presented. The FPGA architecture can update ancestral parsimony vectors regardless of the strategy used to search the tree space and calculates final parsimony scores for tree topologies.

The statistical model of maximum-likelihood estimation in phylogenetics includes the calculation of probabilities for possible tree topologies (evolutionary hypotheses), while Bayesian inference assumes an *a priori* probability distribution of possible trees and employs Markov Chain Monte Carlo sampling algorithms. Both maximum likelihood and Bayesian inference rely on the likelihood function to assess possible tree topologies and explore the tree space. The PLF accounts for approximately 85% to 95% of the total execution time of all likelihood-based phylogenetic tools [182]. Four hardware architectures to compute the PLF are presented. Each design is optimized for a particular purpose. The first two architectures explore different ways to place processing units on reconfigurable logic. In the 1^{st} generation, the processing units are placed in a balanced tree topology, thereby allowing for a deep pipelined datapath and reduced memory requirements. This topology is ideal for fully balanced binary trees. The 2^{nd} generation introduces a vector-like placement of processing units. Such a configuration is generally desirable since performance of the design is independent of the tree topology and search strategy. The 3^{rd} generation is a complete PLF co-processor that comprises dedicated components for all compute-intensive functions of the widely used program RAxML [182]. In addition, it supports all possible input data types: binary, DNA, RNA secondary structure, and protein data. Finally, the 4^{th} generation architecture is a DNA-optimized version of the 3^{rd} generation components for the computation of ancestral probability vectors.

As far as large-scale phylogenomic analyses with hundreds or even thousands of genes are concerned, a property that characterizes the input datasets is that they tend to be gappy.

This means that the MSAs contain taxa with many and disparate missing genes. Currently, in phylogenomic analyses there are alignments missing up to 90% of data. A generally applicable mechanism that allows for reducing the memory footprints of likelihood-based phylogenomic analyses in proportion to the amount of missing data in the alignment is presented. Additionally, a set of algorithmic rules to efficiently conduct tree searches via subtree pruning and regrafting moves using this mechanism is introduced and implemented in RAxML.

In the context of simultaneous sequence alignment and tree building, novel algorithms have recently been introduced to align short sequence reads to reference alignments and corresponding evolutionary trees. PaPaRa [35] is a dynamic programming algorithm for this purpose. An efficient way to boost performance of sequence alignment programs using hardware accelerators is to offload the alignment kernel (dynamic programming matrix calculations) to the accelerator while maintaining the execution of the trace-back step on the CPU. The alignment kernel of PaPaRa has been accelerated using FPGAs and GPUs. The respective reconfigurable architecture is presented. Also, a series of effective optimizations to port the alignment kernel to SIMT (Single Instruction, Multiple Threads) architectures is described.

In population genetic studies, statistical tests that rely on the selective sweep theory [127] are used to identify targets of recent and strong positive selection by analyzing single nucleotide polymorphisms in intra-species MSAs. A recently introduced method to identify selective sweeps is the ω statistic [105]. It uses linkage-disequilibrium (LD) information to capture the non-random association of alleles or states at different alignment positions. Selective sweep theory predicts a pattern of excessive LD in each of the two genomic regions that flank an advantageous and recently fixed mutation. A high-performance approach to compute the ω statistic is described. Additionally, a fast technique to calculate sums of LD values in a genomic region based on a dynamic programming algorithm is introduced. OmegaPlus, a software tool that implements these computational advances has been made available under GNU GPL. The tool can rapidly process extremely large datasets without requiring excessive amounts of main memory, therefore allowing for using off-the-shelf computers to carry out whole-genome analyses.

The scientific results of this thesis have been published in seven peer-reviewed conference papers [11, 14, 15, 18, 21, 32, 185]. One more conference paper [13] has been accepted for publication and two journal articles [12, 20] are currently under minor revision. Additionally, three conference papers [10, 16, 19] and a journal article [17] have been published, which describe FPGA cores that were employed in the implementation and/or verification of the reconfigurable architectures. The papers are available for download in PDF format at `http://www.exelixis-lab.org/publications.html`.

1.3 Structure of the thesis

The remainder of this thesis is organized as follows: Chapter 2 introduces the basic concepts and methods for inferring phylogenetic trees from molecular sequence data. Chapter 3 pro-

vides a short introduction to FPGAs, GPUs, and respective programming models. Chapter 4 presents a reconfigurable system for the acceleration of the phylogenetic parsimony kernel. The AVX vectorization (Section 4.4.3) was developed in collaboration with Alexandros Stamatakis. Chapter 5 presents four reconfigurable architectures for computing the phylogenetic likelihood function. The host-side management (Section 5.5.4) was designed and tested in collaboration with Simon A. Berger. Chapter 6 describes the acceleration of a phylogeny-aware short read alignment kernel using reconfigurable hardware, graphics processing units, and vector intrinsics. The SIMD implementation (Section 6.4.2) was developed in collaboration with Simon A. Berger. Chapter 7 introduces an efficient algorithmic approach to search the tree space on phylogenomic alignments with missing genes using maximum likelihood. Chapter 8 introduces the basic concepts of selective sweep detection and presents an optimized software implementation to compute the ω statistic on whole-genome datasets. Finally, Chapter 9 provides a conclusion and addresses directions of future work.

Chapter 2

Phylogenetic Tree Inference

This chapter introduces the basic concepts of sequence alignment and phylogenetic tree inference. Furthermore, it describes the most commonly used alignment algorithms and tree reconstruction methods. Finally, it provides an overview of related state-of-the-art tools.

2.1 Introduction

Living organisms inhabit every habitable corner of the Earth, from the poles to the equator, from freezing waters to dry valleys, to several miles in the air, and to groundwater thousands of feet below the Earth's surface. Over the last 3.7 billion years, the living organisms have diversified and adapted to nearly every environment. The Greek philosopher Aristotle (384–322 BC) was the first to carry out a classification of species. He classified all living organisms known at that time as either plants or animals. Today, scientists have identified approximately 2 million species on Earth, which are divided into three groups based on their genetic similarity: Bacteria, Archaea, and Eukarya. These three groups are called domains of life. Figure 2.1 illustrates the three domains of life.

All known organisms today, living and extinct, are connected through descent from a common ancestor. This intuitive connection of organisms is mostly depicted as a bifurcating (binary) tree structure that represents their evolutionary history. Evolutionary trees, also known as phylogenetic trees or phylogenies, clearly show the differences between organisms. Evolutionary relationships among species are derived based upon similarities and differences in their morphological and/or genetic characteristics. The evolutionary history of all life forms on Earth—usually referred to as the Tree of Life—is of interest to scientists for the sake of knowledge. However, small-scale phylogenetic trees that comprise only some groups of species find application in several scientific and industrial fields. D. Bader *et al.* [27] present a list of interesting industrial applications of phylogenetic trees such as in vaccine [83] and drug [42] development or for studying the dynamics of microbial communities [60]. Phylogenies are also used to predict the evolution of infectious diseases [43] or the functions of uncharacterized genes [58]. Forensics [148] is one more field that makes use of phylogenetic trees.

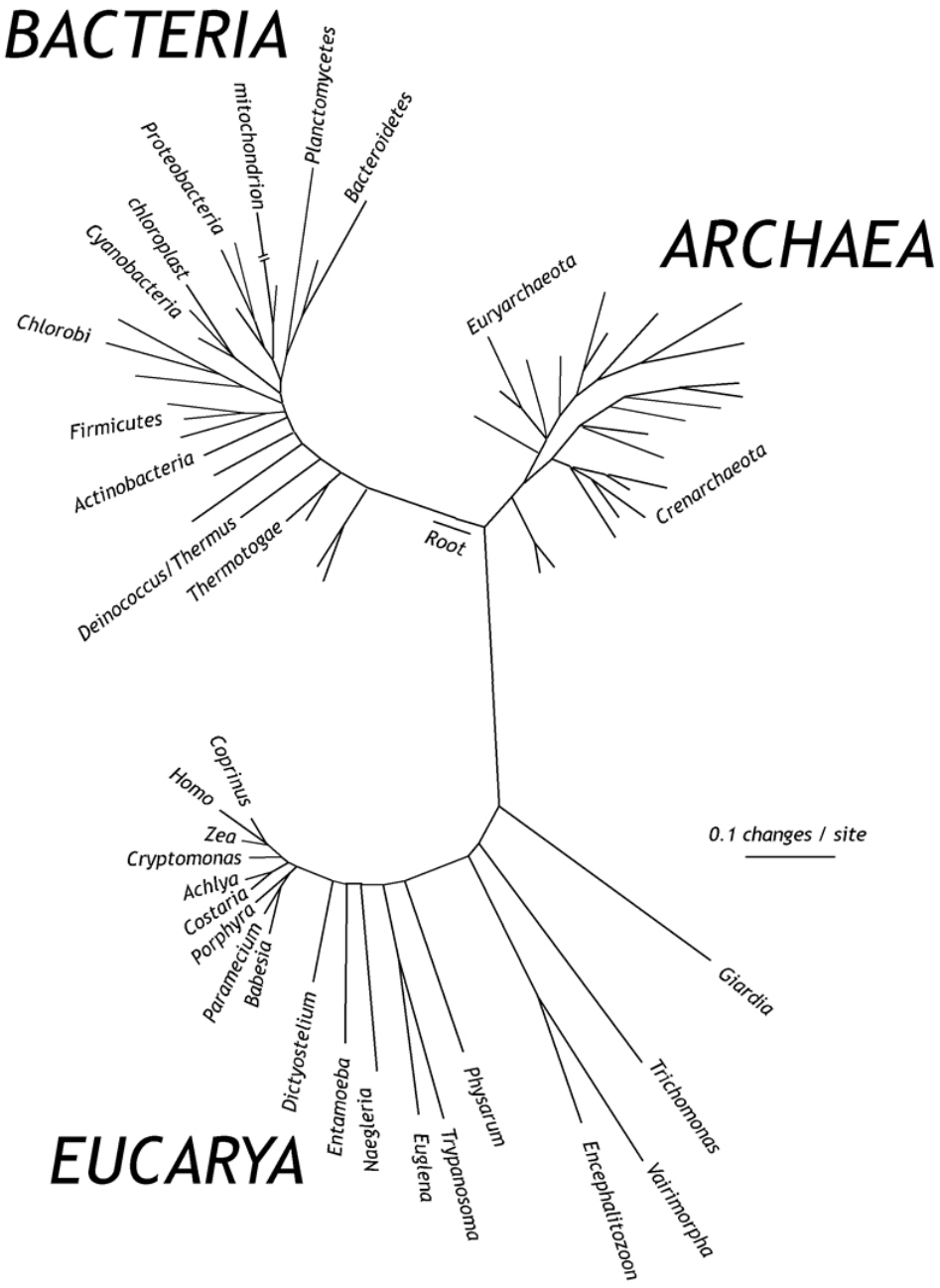


Figure 2.1: Tree showing the three domains of life (Bacteria, Archaea, Eucarya) based on sequencing of 16S ribosomal RNA. (Source: http://www.landcareresearch.co.nz)

Phylogenetic tree reconstruction is based on the assumption that present-day living organisms are the result of diversification and evolution from ancestral populations or genes. All living organisms can replicate and the replicator molecule is the DNA. DNA (deoxyribonucleic acid) is the genetic material of all known living organisms (apart from RNA viruses). Segments of DNA, which are called genes, carry genetic information used in the development and functioning of the organisms. The information in the DNA is stored as a code consisting of four chemical bases: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T). Each base is attached to a sugar molecule as well as a phosphate molecule, and all together form a nucleotide. Similar to the way letters are placed in a certain order to form words, the information available for building and maintaining an organism is determined by the order of nucleotide bases. DNA is organized into a double helix as shown in Figure 2.2.

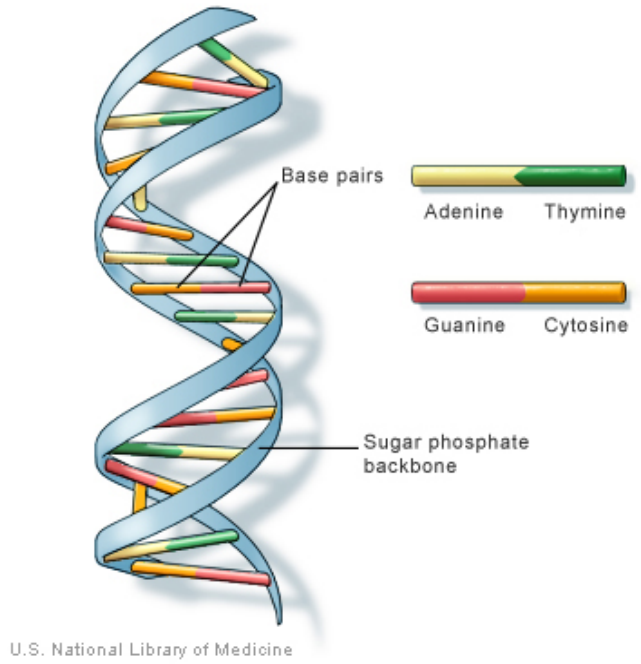


Figure 2.2: DNA is a double helix formed by base pairs attached to a sugar-phosphate backbone. (Source: http://ghr.nlm.nih.gov/handbook/basics/dna)

As previously mentioned, DNA sequences are the prerequisite for all genetic analyses. Raw molecular sequence data are produced by DNA sequencing machines that analyze light signals originating from fluorochromes which are attached to nucleotide bases. Sequencing methods determine the order of nucleotide bases in a DNA molecule. The first reliable DNA sequencing attempts date back to the early 1970s when Frederick Sanger developed the chain-termination method. Since then, DNA sequencing has become several

orders of magnitude faster and cheaper due to next-generation, high-throughput technologies such as pyrosequencing (`http://www.pyrosequencing.com`), Illumina (Solexa) sequencing (`http://www.illumina.com/technology/solexa_technology.ilmn`), DNA nanoball sequencing (`http://www.completegenomics.com`) and others. Nowadays, sequencing machines generate millions of DNA sequences that can be some hundreds of bases long depending on the technology. To proceed with downstream analyses, a multiple sequence alignment (MSA) is required, that is, a sequence alignment of three or more biological sequences (protein, DNA, or RNA) which share a lineage (common evolutionary history). This MSA can then be used in phylogenetic analyses to infer the evolutionary history of the sequences.

2.2 Sequence alignment

Sequence alignments of DNA or protein data represent the usual starting point for phylogeny reconstruction. Higher-level information, like the order of genes in the genome, can also be used. Note however that gene order phylogenetic inference [131] is more computationally intensive than alignment-based inference and, therefore, less suitable to accommodate today's molecular data flood. Sequence alignment algorithms aim to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [133]. In phylogenetic studies, the quality of the final result can only be as good as the quality of the alignment. Thus, a "good" alignment of sequences is the most important prerequisite to conduct a phylogenetic analysis [180].

Alignment approaches are divided into two categories based on the optimization criterion: global optimization or local optimization. Global optimization approaches align two sequences forcing the alignment to span the entire length of the sequences. Local alignment algorithms on the other hand search for subsequences of high similarity. The result is an alignment of only those parts of the sequences that match well. In the following, we provide a brief description of the most popular algorithms and tools for pairwise and multiple sequence alignment.

2.2.1 Pairwise alignment

Pairwise alignment algorithms are used to compute the local or global alignment of only two sequences. There are three primary computational approaches to produce pairwise alignments: dot-matrix, dynamic programming, and word methods [133]. The most known pairwise alignment algorithms are the Needleman-Wunch algorithm [135] for producing global alignments and the Smith-Waterman algorithm [175] for producing local alignments. The Needleman-Wunch algorithm was published in 1970 by Saul B. Needleman and Christian D. Wunsch. It was the first application of dynamic programming to biological sequence comparison. In 1981, Temple F. Smith and Michael S. Waterman proposed the Smith-Waterman algorithm, a variation of the Needleman-Wunch algorithm to find the optimal local alignment of two sequences given a scoring system. Both algorithms exhibit computational similarities since they rely on dynamic

programming. For this reason, only an example for the Smith-Waterman algorithm is given here.

To compare two sequences $A = (a_1a_2a_3...a_n)$ and $B = (b_1b_2b_3..b_m)$, the Smith-Waterman algorithm initializes the first row and column of the dynamic programming matrix with zeros:

$H_{i,0} = 0, 0 \leq i \leq n$
$H_{0,j} = 0, 0 \leq j \leq m$

and uses pairwise comparisons between individual characters to fill the rest of the matrix as follows:

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + s(a_i, b_j) & Match/Mismatch \\ H_{i-1,j} + s(a_i, -) & Deletion \\ H_{i,j-1} + s(-, b_j) & Insertion \\ 0 \end{cases} \quad 1 \leq i \leq n, 1 \leq j \leq m, \quad (2.1)$$

where s is the scoring function: if $a_i = b_j$ then $s(a_i, b_j) = s(match)$, if $a_i \neq b_j$ then $s(a_i, b_j) = s(mismatch)$, and $s(a_i, -) = s(-, b_j) = s(gap)$ is the gap penalty. To construct the local alignment from the matrix, the starting point for tracing back the path is the highest-scoring matrix cell. The trace-back continues until a cell with zero score is found. Each matrix cell score represents the maximum possible score for all alignments that end at the coordinates of the cell. Thus, starting from the cell with the highest score in the matrix yields the optimal local alignment. For example, one can consider the sequences $S_1 = ACCCTTGCT$ and $S_2 = CTGACTT$ as well as a scoring function: $s(match) = 1$, $s(mismatch) = -1$, and $s(gap) = -2$. The scoring matrix is computed as shown in Figure 2.3. The trace-back step starts with the highest value in the matrix and yields the following optimal local alignment:

```
A A C C C T T
- A - - C T T.
```

There exist improved variants of the Smith-Waterman algorithm that allow for better accuracy [25] and reduced complexity [77]. Furthermore, there exists extensive literature on accelerating the algorithm using SIMD (Single Instruction, Multiple Data) instructions on general purpose CPUs, GPUs, and hardware description languages on FPGAs. Chapter 6 provides a brief overview of these high-performance implementations.

Despite the fact that the Smith-Waterman algorithm finds the optimal local alignment via a full alignment procedure, the required dynamic programming approach does not allow to search huge databases of sequences within an acceptable amount of time. To this end, BLAST was introduced in 1990 [26]. BLAST stands for Basic Local Alignment Search Tool and represents a heuristic approximation of the Smith-Waterman algorithm. BLAST compares a query sequence

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

Figure 2.3: Dynamic programming matrix of the Smith-Waterman algorithm.

with a library of sequences and identifies those sequences in the library that have a higher similarity score than a given threshold. It does not compare two sequences entirely but locates initial word matches (seeds) between the sequences and calculates local alignments afterwards. Evidently, BLAST does not guarantee to find the optimal local alignment, but its relatively good accuracy in combination with reduced execution times has established BLAST as the natural choice for huge genome database searches.

2.2.2 Multiple sequence alignment

Multiple sequence alignment (MSA) tackles the problem of aligning three or more sequences. Several computational approaches for generating MSAs exist, e.g., progressive alignment algorithms [142, 198], iterative methods [78], hidden Markov models [94], or genetic algorithms [141]. Constructing MSAs includes two major challenges: i) which scoring function to use and ii) how to find the optimal alignment given a scoring function. Dynamic programming algorithms can be used to align small numbers of homologous sequences [119]. For an alignment of n sequences for instance, the construction of a n-dimensional dynamic programming matrix is required. Clearly, such approach becomes extremely compute- and memory-intensive for large numbers of sequences since the search space increases exponentially with n. Heuristic approaches are deployed to make the alignment of many sequences computationally feasible. The underlying idea of heuristic methods is to determine pairwise alignments of closely related sequences and progressively add other, less related sequences to the alignment. Finding the optimal MSA is a NP-complete problem under most reasonable scoring functions [201].

A variety of MSA algorithms exist, but the problem of MSA construction is beyond the scope of this thesis and, therefore, only the most commonly used algorithms are briefly mentioned. The ClustalW [198] and T-Coffee [142] alignment tools are based on the progressive method.

MUSCLE [54] implements an iterative approach and improves upon accuracy as well as speed compared to ClustalW and T-Coffee. MAFFT [102, 103], which implements a progressive method as well as an iterative refinement procedure, outperforms ClustalW and T-Coffee in terms of speed while exhibiting comparable accuracy. FSA (Fast Statistical Alignment [38]) is based on pair hidden Markov models to approximate an insertion/deletion process on a tree and employs a sequence annealing algorithm to combine the posterior probabilities estimated from these models into a multiple sequence alignment. BAli-Phy [159, 193] is a Bayesian posterior sampler that employs Markov chain Monte Carlo to simultaneously estimate multiple sequence alignments and the phylogenetic trees that relate the sequences. Another tool for joint multiple sequence alignment and tree reconstruction is POY [203]. Detailed reviews of state-of-the-art MSA software tools as well as benchmarking can be found in [55] and [197].

2.3 Evolutionary relationships

Biological diversity can be classified by assessing degrees of apparent similarity and difference among organisms. Classification of organisms based on observable morphological traits assumes that the biological relationship between organisms is closer the greater the degree of physical similarity is. Nowadays, a generally accepted scientific approach to classify organisms is based on shared evolutionary history. The evolutionary relationships of a group of organisms that share an evolutionary history are depicted by phylogenetic trees.

2.3.1 Phylogenetic trees

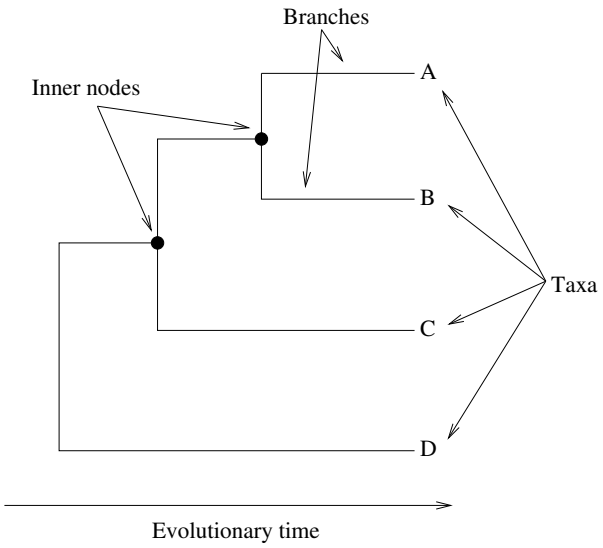


Figure 2.4: A generic phylogenetic tree.

Phylogenetic trees are a graphical way to represent evolutionary relationships. A phylogenetic tree is a tree diagram that consists of leaves (tip nodes) and inner nodes. The tip nodes correspond to individual organisms, species, or sets of species. The items represented by the tip nodes are usually referred to as taxa. The inner nodes (i.e., the branching points within a tree) represent ancestral species. Each inner node corresponds to the most recent common (hypothetical) ancestor of the lineages that descend from that node. The branches of a phylogenetic tree (i.e., the lines that connect all tips and inner nodes) represent the path that traces back the evolutionary history of the lineages. Figure 2.4 shows a generic phylogenetic tree.

Phylogenetic trees can be rooted or unrooted. The root represents the common ancestor of all entities at the leaves of the tree and indicates the direction of the evolutionary process. The term "sister group" is used to describe closely related taxa in rooted trees. Taxa in a sister group share a recent common ancestor, i.e., the inner node that joins them together. Unrooted trees illustrate the relatedness of the entities at the leaves without providing any further information about ancestral relationships. While rooted trees can be trivially transformed into unrooted trees by simply omitting the root, to generate rooted trees based on unrooted ones usually requires the use of an outgroup. An outgroup is a lineage that is known to be more distantly related to all organisms at the tips. Figure 2.5 shows examples of a rooted and an unrooted tree with five taxa. Trees can be either bifurcating or multifurcating. Every inner node in a bifurcating tree has exactly three neighbors (binary tree topology), whereas multifurcating trees allow for more than three neighbors per inner node.

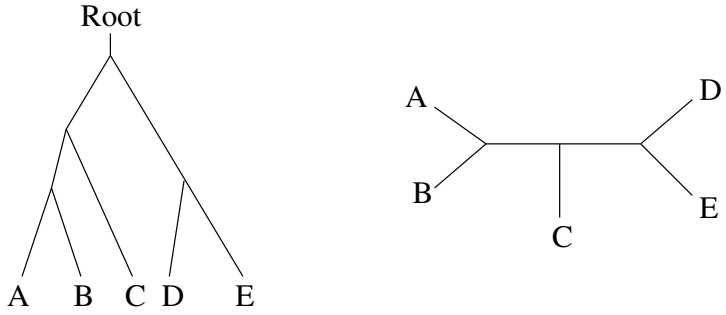


Figure 2.5: A rooted (left) and an unrooted (right) phylogenetic tree with five taxa.

A classification of ogranisms in phylogenetic trees according to the information at the tips leads to two types of trees: species trees and gene trees. Species trees depict evolutionary relationships between species or populations of species. Figure 2.6 illustrates a species tree that shows the divergence of human and ape species. A gene tree (Figure 2.7) on the other hand, is a tree of a group of orthologous genes, each sampled from a different species [136, 196]. Two genes are orthologous if they diverged after a speciation event, i.e., a lineage-splitting event that produces two or more separate species. The outcome of a phylogenetic analysis is a gene tree and does not necessarily represent the actual evolutionary history of the species (species

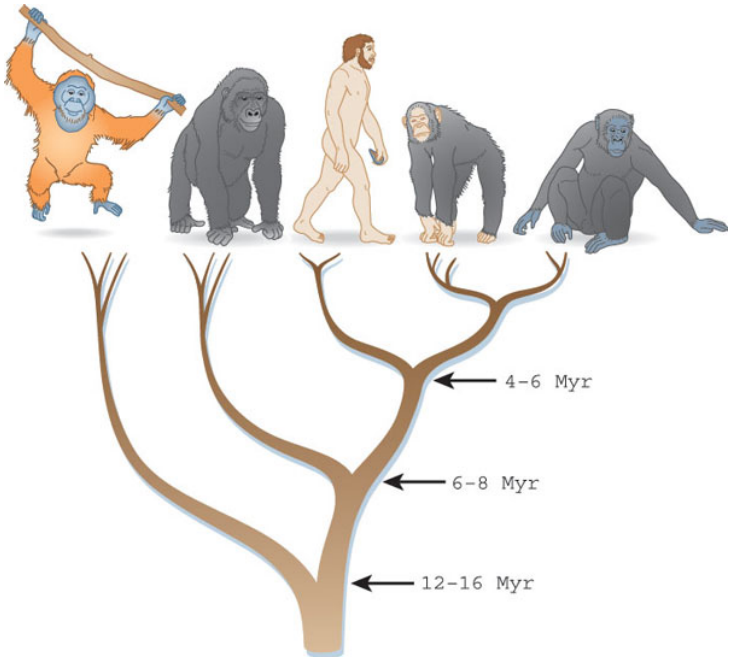


Figure 2.6: A species tree showing the divergence of human and ape species. Approximate dates of divergence are given for, from left to right, orangutan, gorilla, human, bonobo and chimpanzee. (Source: [150])

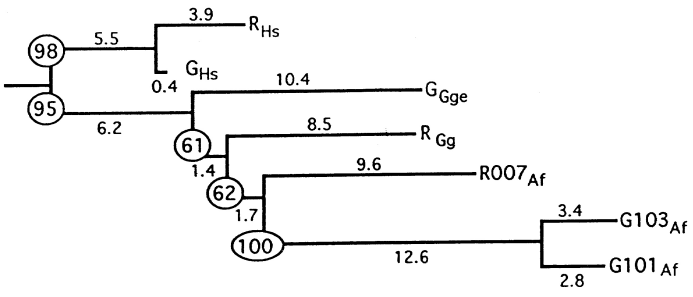


Figure 2.7: Phylogenetic tree of visual pigment genes. R: red-sensitive pigment; G: green-sensitive pigment. Subscripts as follows: Hs–humans; Gg–chickens; Gge–gecko; Af–fish. (Source: [213])

tree), although ideally these two trees should be identical. Several studies [136, 152, 169] have evaluated the probability that the topology of the gene tree is the same as that of the species tree for various cases such as number of species, number of alleles, or sampling errors with respect to the number of nucleotides.

2.3.2 Problem complexity

Phylogenetic tree reconstruction faces a major computational issue: the number of potential alternative tree topologies that need to be evaluated grows super-exponentially with the number of species. For n species for instance, the number of rooted and unrooted binary tree topologies is given as follows [56]:

$$Rooted\ trees \quad = \quad \frac{(2n-3)!}{2^{n-2}(n-2)!} \qquad 2 \leq n \tag{2.2}$$

$$Unrooted\ trees \quad = \quad \frac{(2n-5)!}{2^{n-3}(n-3)!} \qquad 3 \leq n. \tag{2.3}$$

Table 2.1 shows the number of different tree topologies with and without root for up to 100 species. Note that, for a number of species as low as 52, the number of different tree topologies (either rooted or unrooted) is close to the estimated number of atoms in the universe ($\approx 10^{80}$). Thus, it becomes evident that, regardless of the scoring method used to evaluate a tree topology, it is not feasible to evaluate all possible topologies, even for small numbers of species. As a matter of fact, it has been shown that maximum likelihood [46] and maximum parsimony [50]

Species	Rooted trees	Unrooted trees
3	3	1
4	15	3
5	105	15
6	945	105
7	10,395	945
8	135,135	10,395
9	2,027,025	135,135
10	34,459,425	2,027,025
15	$2.13 * 10^{14}$	$7.90 * 10^{12}$
20	$8.20 * 10^{21}$	$2.21 * 10^{20}$
25	$1.19 * 10^{30}$	$2.53 * 10^{28}$
50	$2.75 * 10^{76}$	$2.83 * 10^{74}$
75	$4.09 * 10^{128}$	$2.78 * 10^{126}$
100	$3.34 * 10^{184}$	$1.70 * 10^{182}$

Table 2.1: Number of possible rooted and unrooted trees with 3–100 organisms. (Computed using TreeCounter [179])

methods, which both are commonly used in practice, are NP-hard problems. This is the reason that heuristic approaches are deployed in order to search the tree space as efficiently as possible to find the best possible, yet suboptimal topology given a scoring criterion. Furthermore, it has been demonstrated [80, 205] that, while fast scoring methods allow for a more exhaustive search of the tree space, more elaborate scoring functions such as maximum likelihood lead to the detection of better trees. Thus, there is an apparent trade-off between the execution time of an analysis and the quality of the outcome.

2.4 Phylogenetic inference methods

There exist two main classes of phylogeny reconstruction methods: distance-matrix methods and character-based methods. Distance-matrix methods, such as UPGMA [176] (Unweighted Pair-Group Method with Arithmetic Mean) or neighbor joining [168], reconstruct a phylogeny of n species by computing a $n \times n$ distance matrix which contains all pairwise distances between the n sequences. Character-based methods on the other hand, such as maximum parsimony [68] and maximum likelihood [65], use the input MSA to compute tree scores on a column-by-column basis. Distance-matrix methods are much faster but also less accurate than character-based methods, which explains why they are mostly deployed to obtain initial estimates of phylogenetic trees.

The accuracy of phylogenetic inference methods is usually assessed using simulated data. Typically, a tree (true tree) and an evolutionary model are used to simulate sequences. These simulated sequences are then provided as input to phylogenetic tools to infer trees. The accuracy of the employed methods is assessed by examining the topological distance of the inferred trees from the true tree. Although distance-matrix methods represent the only computationally feasible approach to reconstruct very large phylogenies, several studies have revealed that character-based methods usually recover a tree that is topologically closer to the true tree [92, 110, 166]. This section provides a brief description of the most commonly used character-based methods.

2.4.1 Maximum parsimony

Maximum parsimony (MP) strives to find the phylogenetic tree that explains the evolutionary history of a set of organisms by the least amount of evolutionary changes. Thus, the most parsimonious tree is considered the one that has been obtained with the least number of mutations. Since the parsimony criterion is a character-based method, it operates on the input MSA on a column-by-column basis. Note however that not all columns (sites) in the MSA are informative. Informative sites are those that consist of at least two different character states, and each of these states must occur in more than one sequences. Figure 2.8 shows an example of an MSA with four sequences. Only three out of the eight sites in the MSA are informative and can, therefore, be used in the analysis. Parsimony-uninformative sites are excluded from the analysis because they have the same score regardless of the tree topology. Figure 2.8 also

depicts the three unrooted tree topologies for the given number of sequences and shows the parsimony scores.

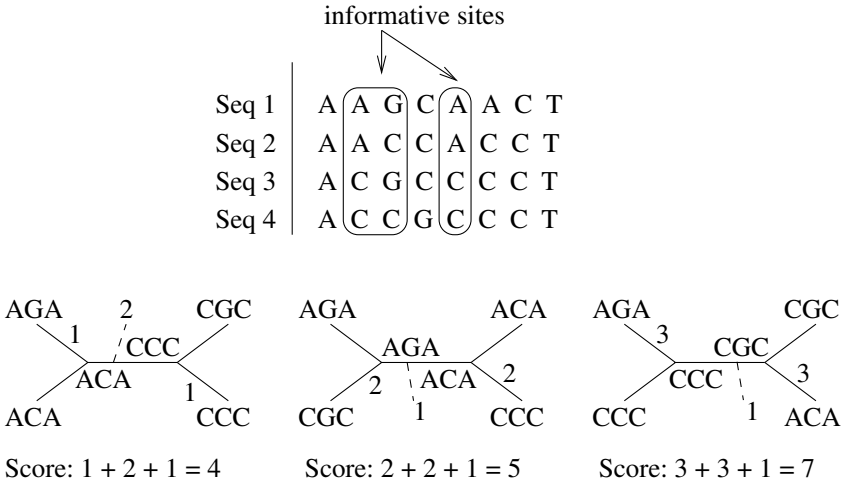


Figure 2.8: Calculation of parsimony scores for the alternative unrooted tree topologies based on the parsimony informative sites of the 4-sequence MSA and an arbitrary rooting.

MP is not statistically consistent due to the phenomenon of long branch attraction (LBA) [64, 90]. Given an infinite amount of input data, there is no guarantee that MP will recover the true tree. LBA can be observed when very long branches are inferred to be closely related though they have evolved from very different lineages. Similarity between sequences on long branches may be explained by independent multiple substitutions/mutations of the same nucleotide and not by their close relationship. Joe Felsenstein [64] showed the statistical inconsistency of the parsimony criterion in 1978, and this is the reason that the situation in which this inconsistency occurs is also called the "Felsenstein zone".

2.4.2 Maximum likelihood

Maximum likelihood (ML) is a statistical method to fit a mathematical model to the data at hand. According to the likelihood principle, the preferred tree topology is the one that maximizes the probability of observing the data, i.e., the input MSA. The first application of ML in phylogenetic tree reconstruction dates back to 1964 and is due to A.W.F. Edwards and L.L. Cavalli-Sforza [56]. In 1981, J. Felsenstein [65] fully developed a general ML approach for nucleotide sequence data. The method was also applied on protein data by Kishino in 1990 [109] as well as by Adachi and Hasegawa in 1992 [9]. In addition to the fact that ML is statistically consistent [162], it is also considered to be less error-prone than every other method since it is not considerably affected by errors during the sequencing process.

A ML phylogenetic analysis requires an evolutionary model, that is, a probabilistic model of nucleotide substitution which explains the differences between two sequences. Given a hy-

pothesis Φ and observed data Δ, the likelihood of the data is given by $L(\Delta;\Phi) = P(\Delta|\Phi)$, which is the probability of obtaining Δ given Φ. Note that the terms 'likelihood' and 'probability' are two distinct concepts when used in a statistical context. Probability represents the odds of observing an unknown outcome based on known input parameters, whereas likelihood is used to predict unknown parameters based on a known outcome. In a phylogenetic context, the unknown hypothesis Φ is the phylogenetic tree and the evolutionary model (including the parameters of the model), while the known data Δ is the genetic material of the species under investigation (DNA sequences in the input MSA). Thus, the likelihood $L(\Delta;\Phi)$ of a tree describes how likely it is that the MSA was generated by the specific tree and evolutionary model.

Apart from devising a model of molecular evolution and deploying the likelihood function as optimality criterion to score tree topologies, one more problem remains to be resolved: how to search for the maximum likelihood tree. As pointed out in Section 2.3.2, the number of possible tree topologies grows exponentially with the number of taxa. This makes the creation and evaluation of all possible tree topologies impossible even for a small number of sequences due to prohibitively long execution times. To resolve this issue and make the reconstruction of a phylogeny that comprises hundreds or even thousands of organisms feasible, heuristic approaches are employed to search the tree space. Since heuristic tree-search strategies do not evaluate all possible topologies, the outcome of an analysis is a tree that represents a trade-off between topological accuracy and execution time.

2.4.2.1 Nucleotide substitution models

As already mentioned, a ML analysis relies on a nucleotide substitution model which describes the process of one nucleotide evolving into another. The process of base substitution within a population is generally so slow that it can not be observed during the lifetime of a human being, but a detailed examination of DNA sequences can provide insight on the process. Evolutionary changes in DNA sequences are revealed by comparisons between species that share a common ancestor. These comparisons require the use of statistical methods that make several assumptions about the nucleotide substitution process. These methods usually assume that individual sequence sites have evolved independently of each other. The change from a character state i to a character state j along a branch of a tree is usually modeled as a continuous-time homogeneous Markov process. The main property of a Markov chain is that it is memoryless. This means that the change from a state A to a state B depends only on the current state A. To put this into an evolutionary context, assume a sequence s and the nucleotide base A at a position x of the sequence at time t_0. The probability of observing another nucleotide base, T for instance, at the same position at some later time t_1 depends only on the fact that an A was present at time t_0.

The mathematical formulation of an evolutionary model is a matrix of base substitution rates. There are 4 possible character states in a nucleotide sequence (A, C, G, T for DNA data

and A, C, G, U for RNA data), while there are 20 possible character states (A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y) in an amino acid sequence. Thus, the base substitution matrix is of size 4×4 when it models nucleotide substitution rates whereas it is of size 20×20 when it models amino acid substitution rates. The generic form of the nucleotide substitution matrix is given by Equation 2.4. Each Q_{ij} element represents the substitution rate from a base i to a base j in an infinitely small time interval dt.

$$Q = \begin{pmatrix} -\mu(a\pi_C + b\pi_G + c\pi_T) & \mu a\pi_C & \mu b\pi_G & \mu c\pi_T \\ \mu g\pi_A & -\mu(g\pi_A + d\pi_G + e\pi_T) & \mu d\pi_G & \mu e\pi_T \\ \mu h\pi_A & \mu j\pi_C & -\mu(h\pi_A + j\pi_C + f\pi_T) & \mu f\pi_T \\ \mu i\pi_A & \mu k\pi_C & \mu l\pi_G & -\mu(i\pi_A + k\pi_C + l\pi_G) \end{pmatrix} \quad (2.4)$$

In this matrix, the rows and columns are ordered by the alphabetical order of the bases A, C, G, and T. Parameter μ represents the mean instantaneous substitution rate from one nucleotide to another. The parameters a, b, ..., l correspond to every possible transformation between distinct bases. The product of each one of them and the mean instantaneous substitution rate constitutes a rate parameter. Parameter μ is usually set to 1, and parameters a, b, ..., l are scaled such that the average substitution rate is 1. Parameters π_A, π_C, π_G, and π_T are called base frequencies and represent the frequency of occurrence of the bases A, C, G, and T in the input MSA. It is assumed that the nucleotide substitutions are at equilibrium, that is, the proportions of the individual bases remain constant over time. Thus, they also serve as the stationary distribution of the Markov process. The diagonal elements are chosen such that all elements in each row sum up to zero ($Q_{ii} = -\sum_{i \neq j} Q_{ij}$), since the rate $-Q_{ii}$ that the Markov chain leaves state i must be equal to the rate $\sum_{i \neq j} Q_{ij}$ of arriving at all other states together. All models used in real-world analyses are derived from matrix 2.4.

A widely used assumption is that the substitution rate from a base i to a base j in a given period of time Δt is the same as the substitution rate from base j to base i. Evolutionary models that respect this assumption are called *time-reversible* and utilize time-reversible Markov chains. The assumption of time reversibility leads to the following equations for the parameters a, b, ..., l: $g = a$, $h = b$, $i = c$, $j = d$, $k = e$ and $l = f$. Applying these equations on matrix 2.4 and setting $\mu = 1$ leads to the simplified matrix 2.5, which represents the most general form of Q for time reversibility. The evolutionary model described by this matrix is known as the General Time-Reversible (GTR) model [111, 161]. Although there is no biological reason why nucleotide substitution models should be reversible, time-reversibility is mathematically convenient. Furthermore, some of the time-reversible models come close enough to fit real data.

$$Q = \begin{pmatrix} -(a\pi_C + b\pi_G + c\pi_T) & a\pi_C & b\pi_G & c\pi_T \\ a\pi_A & -(g\pi_A + d\pi_G + e\pi_T) & d\pi_G & e\pi_T \\ b\pi_A & d\pi_C & -(h\pi_A + j\pi_C + f\pi_T) & f\pi_T \\ c\pi_A & e\pi_C & f\pi_G & -(c\pi_A + e\pi_C + f\pi_G) \end{pmatrix} \quad (2.5)$$

Some models impose additional restrictions to the parameters of matrix Q by reducing the number of rate parameters and/or the number of base frequencies. The JC69 [98] model for

instance, proposed by Jukes and Cantor, assumes equal base frequencies ($\pi_A = \pi_C = \pi_G = \pi_T = 0.25$) and rate parameters ($a = b = c = d = e = f = g = h = i = j = k = l$). Other restricted models are the F81 model by Felsenstein [65] with equal rate parameters and 2 base frequencies, the K2P model by Kimura [108] with 2 rate parameters and equal base frequencies, and the HKY85 model by Hasegawa, Kishino, and Yano [85] with 2 rate parameters and 4 distinct base frequencies. Note that these models are considered as being oversimplified. As a result, the GTR model is predominantly used in modern phylogenetic analyses [160].

Matrix Q provides the substitution rates between bases in an infinitely small time interval dt. However, a substitution probability matrix P is required in order to compute the likelihood of a tree. Each p_{ij} element in P represents the probability that a base i will evolve into base j after time t. The substitution probability matrix P is computed as shown in Equation 2.6.

$$P(t) = e^{Qt} \tag{2.6}$$

The exponential can be evaluated by decomposition of Q into its eigenvalues and eigenvectors [114]. There exist closed-form analytical solutions for the substitution probability matrices of the simpler models like JC69, K2P, and HKY85 [195]. For the GTR model, the decomposition of Q requires linear algebra approaches (eigenvalue/eigenvector decomposition) [195].

2.4.2.2 Rate heterogeneity

The aforementioned models assume that all sites of the alignment evolve at the same rate. However, there is strong biological evidence for rate variation among sites. Assuming rate homogeneity among sites represents an oversimplification of the actual evolutionary process. Analyzing alignments that comprise several genes for instance, requires not only different substitution rates among the sites but also a different base substitution model for each gene. Several studies [41, 70, 211] have shown that ignoring rate variation among sites in a likelihood-based phylogenetic analysis can lead to erroneous results if the rates vary among the sites.

Rate heterogeneity among alignment sites can be accommodated by using an additional rate parameter r_i ($i = 1, ..., l$, with l being the number of alignment sites) to compute the substitution probability matrix P as shown in Equation 2.7.

$$P(t, r_i) = e^{r_i Qt} \tag{2.7}$$

Note that the substitution rate matrix Q is fixed, whereas only parameter r varies from site to site leading to a different substitution probability matrix P per site. G. Olsen developed a software tool (DNArates [145]) that computes a ML estimate of the individual per-site rates for a fixed tree topology. However, employing individual per-site evolutionary rates is not often used in real-world analyses because of statistical concerns regarding over-parameterization and over-fitting of the data [181]. Modern phylogenetic software tools account for rate heterogeneity among sites using either the Γ model [209] of rate heterogeneity or, a memory- and time-efficient approximation for it, the CAT model [181].

The Γ model, proposed by Z. Yang in 1993, allows continuous variability of mutation rates over sites. The Γ density function is

$$g(r;\alpha,\beta) = \frac{\beta^{\alpha} r^{\alpha-1} e^{-\beta r}}{\Gamma(\alpha)}. \tag{2.8}$$

A mean distribution rate of 1 is obtained by setting the scale parameter β equal to the shape parameter α, and thus maintain Q's mean substitution rate of 1. The α parameter is inverse to the coefficient of variation of the substitution rate, that is, a low α suggests significant rate differences among sites while a high α means low rate variation. When $\alpha \to \infty$, the distribution degenerates into a model of a single rate for all sites.

Z. Yang also proposed the discrete Γ model [210], which approximates the continuous rate distribution by dividing the sites into distinct rate categories. Yang found that the discrete Γ model with 4 discrete rates provides a good approximation and significantly reduces runtime. Consequently, most software tools for phylogenetic inference employ the discrete Γ model with 4 distinct rate categories instead of the continuous Γ model.

2.4.2.3 Phylogenetic likelihood function

The phylogenetic likelihood function (PLF) computes the likelihood on a given tree topology. All likelihood-based programs deploy the PLF to score trees. The calculation of the PLF is based on two computational techniques, the Felsenstein's pruning algorithm and the Pulley Principle [65], which allow the construction of an iterative procedure for evaluating a fixed/given tree topology. Phylogenetic trees under ML are unrooted for mathematical and computational reasons [65]. According to the Pulley Principle, any branch of an unrooted tree can be regarded as containing the root as long as the Markov process of base substitution is reversible and there are no constraints on the branch lengths. This essentially permits the trivial transformation of an unrooted tree topology into a rooted one by simply placing a *virtual root* into any branch of the unrooted tree. An important feature of the Pulley Principle is that the likelihood score remains the same regardless of where in the tree the *virtual root* is placed. When a *virtual root* has been placed in a tree, the pruning algorithm can be applied from the tips toward the *virtual root* to "prune" the tree and compute the likelihood score. The pruning algorithm is closely related to the peeling algorithm introduced in [59]. Felsenstein gave the name 'pruning algorithm' because every step of the algorithm prunes (removes) two tips from the tree. Figure 2.9 illustrates how a 4-taxon tree topology is pruned according to this algorithm.

The PLF is applied to a fixed rooted tree topology including fixed branch lengths and given model parameters. The nucleotide or amino acid sequences assigned to the tips of the tree are called OTUs (Operational Taxonomic Units), while the inner nodes are called HTUs (Hypothetical Taxonomic Units), since they represent extinct species for which no sequence data are available. The PLF computations start from the tips and proceed, following a post-order tree traversal, toward the *virtual root*. For every alignment site i, at each node k, the conditional probability $L_{s_k}^{(k)}(i)$ is calculated, which describes the probability of observing the

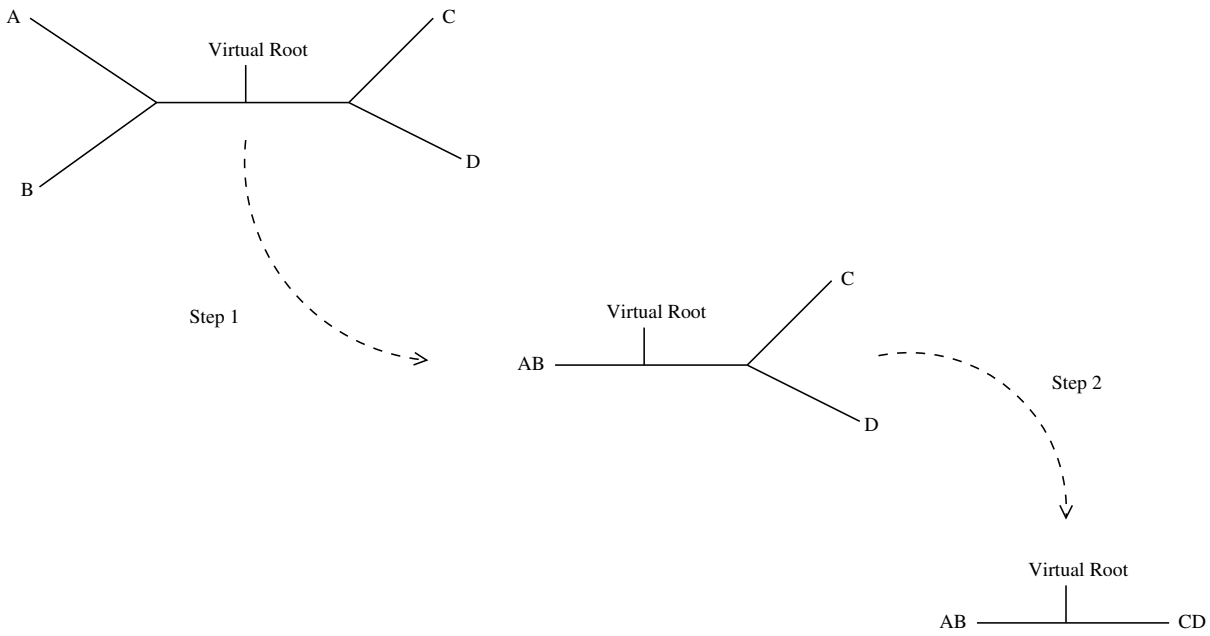


Figure 2.9: Pruning steps for a 4-taxon tree according to Felsenstein's pruning algorithm.

data at the descendants of node k given the state s_k at node k. For the simplest case of applying the PLF on DNA data there are four possible nucleotide states: $s_k \in \{A, C, G, T\}$. Thus, a conditional probability vector $\vec{L}^{(k)}(i)$ (also referred to as likelihood vector) containing four conditional probabilities $L_{s_k}^{(k)}(i)$ (one for each possible nucleotide state) is computed for the alignment site i of node k: $\vec{L}^{(k)}(i) = [L_A^{(k)}(i), L_C^{(k)}(i), L_G^{(k)}(i), L_T^{(k)}(i)]$. The final conditional probability vector $\vec{L}^{(k)}$ for node k contains m vector entries $\vec{L}^{(k)}(i)$, $i = 1...m$, with m being the number of sites in the MSA. Since all employed base substitution models assume that sites evolve independently (see Section 2.4.2.1), the conditional probabilities $\vec{L}^{(k)}(i)$ are computed site-by-site by iterating over the MSA alignment columns. In the following, we describe the process of computing the conditional probabilities and the likelihood score for a single site.

Figure 2.10 provides a schematic representation of a single "pruning" step that removes two tips from the tree. The probabilities at the tips, for which observed data (DNA sequences) is available, are set to 1.0 for the observed nucleotide characters and to 0.0 for all remaining characters at the respective position i. For this example, base G at position i of node l (left child) leads to $P(G) := 1.0$ and $P(A) := P(C) := P(T) := 0.0$. Apart from the standard nucleotide bases A, C, G, and T, there also exist ambiguous characters like M for instance, which stands for A or C and is thus represented by $P(A) := 1.0, P(C) := 1.0, P(G) := P(T) := 0.0$ at the tip vector level.

A branch length represents the evolutionary time between two nodes in the tree in terms of expected substitutions per site. The branch lengths b_l and b_r of the two branches that lead

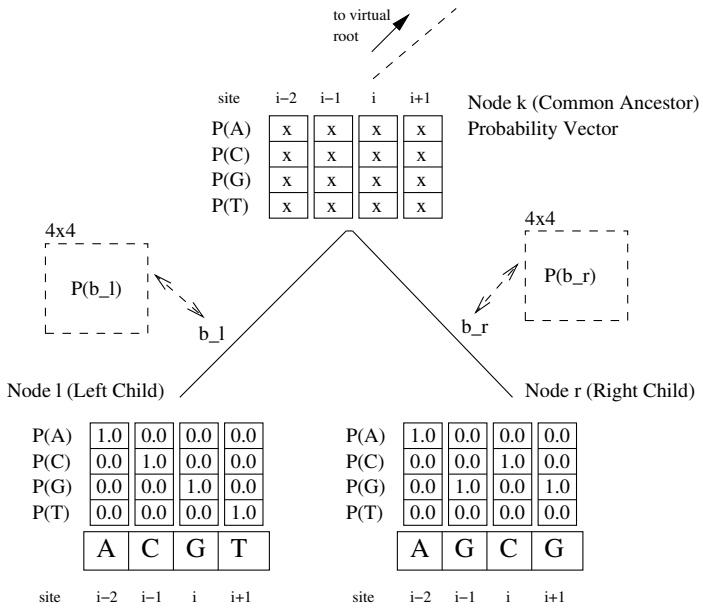


Figure 2.10: A single pruning step that removes two tips from the tree.

to the left and right child nodes are fixed. Given fixed branch lengths and, therefore, fixed substitution probability matrices $P(b_l)$ and $P(b_r)$ (the base substitution model Q is also fixed), the entries of the conditional probability vector at node k are computed as follows:

$$\vec{L}_X^{(k)}(i) = \big(\sum_{S=A}^{T} P_{XS}(b_l)\vec{L}_S^{(l)}(i)\big)\big(\sum_{S=A}^{T} P_{XS}(b_r)\vec{L}_S^{(r)}(i)\big). \tag{2.9}$$

Equation 2.9 calculates the conditional probability of observing a character X ($X \in \{A, C, G, T\}$) at position i of the ancestral node k given character states S ($S \in \{A, C, G, T\}$) at the same position i of the child nodes l and r. When the procedure reaches the *virtual root*, Equation 2.9 computes the conditional probability vector $\vec{L}^{(vr)}$. The likelihood score of site i is computed from the conditional probability vector at the virtual root as follows:

$$l(i) = \sum_{S=A}^{T} \pi_S {L_S}^{(vr)}(i), \tag{2.10}$$

where probabilities π_A, π_C, π_G, and π_T are the *a priori* probabilities (see Section 2.4.2.1) of observing A, C, G, or T at the *virtual root*. These probabilities can all be set to 0.25, be obtained empirically by counting the occurrences of A, C, G, and T in the MSA, or be estimated using maximum likelihood. The final likelihood score of the tree is the product of all m per-site likelihood scores:

$$L = \prod_{i=1}^{m} l(i). \tag{2.11}$$

Note that, because the values in the substitution probability matrices P are ≤ 1, the individual $L_{s_k}^{(k)}(i)$ values can become very small, thus leading to very small per-site likelihood scores $l(i)$. To prevent numerical underflows, the log likelihood score is usually calculated instead of L:

$$LH = \log(L) = \sum_{i=1}^{m} \log(l(i)). \tag{2.12}$$

2.4.2.4 Branch length and model parameter optimization

ML analyses aim to reconstruct the phylogenetic tree that maximizes the likelihood score. To obtain the maximum likelihood score for a fixed tree topology, one needs to optimize all branch lengths *and* all model parameters. Both the branch lengths and the model parameters are considered free parameters of the likelihood function.

Model parameter optimization encompasses the estimation of rate parameters in the substitution rate matrix and the parameters that model among-site rate heterogeneity. Substitution models such as HKY85 [85] or GTR [111, 161] assume that branch lengths and model parameters are not correlated. A numerical optimization algorithm [212] that works well in this case typically exhibits two phases. In the first phase, some model parameters can be updated simultaneously using, for instance, the Broyden-Fletcher-Goldfard-Shanno algorithm (BFGS [69, 72]), while branch lengths remain fixed/constant. In the second phase, branch lengths are updated one by one while the model parameters remain fixed. Since model parameter optimization is computationally expensive, model parameters are optimized only crudely in the initial stages of a tree search. When there is a strong correlation between branch lengths and other model parameters, as is the case with the Γ model [209] of rate heterogeneity, the first phase of the algorithm needs to be embedded into the second phase. Therefore, a multivariate optimization algorithm (such as BFGS) can be used to estimate the model parameters while optimizing branch lengths for any given values of the model parameters.

Modern phylogenetic software tools search the tree space via frequent removal and reinsertion of subtrees or related techniques of tree alteration. The branch lengths need to be optimized after each change to the tree topology. The branch lengths of a tree are usually optimized one by one. All other branch lengths except for the one to be optimized are considered to be fixed. Figure 2.11 shows a branch b of length bl that connects two nodes n_1 and n_2. To optimize this branch, a *virtual root* is initially placed onto it (Pulley Principle [65], see Section 2.4.2.3). The likelihood score of the tree depends on the distances bl_1 and bl_2 between the *virtual root* vr and the two nodes n_1 and n_2. Placing the *virtual root* right next to node n_1 for instance ($bl_1 = 0$) leads to $bl = bl_2$, which allows to optimize the likelihood score of the tree by altering only one parameter (bl_2). The *Newton-Raphson* optimization procedure [156] is often used for this task

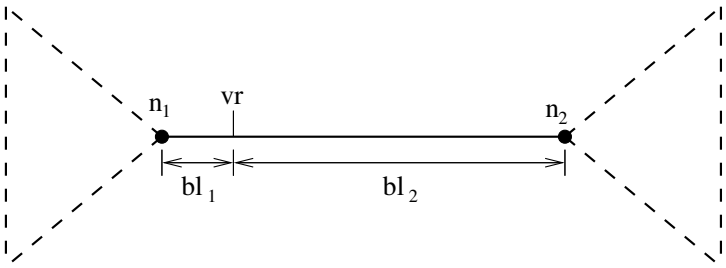


Figure 2.11: A branch of a tree to be optimized.

because it has good convergence properties. As with most univariate optimization methods, the first and second partial derivatives of the PLF with respect to the branch length parameter *bl* need to be calculated.

2.4.3 Bayesian inference

Bayesian inference of phylogenies employs the same substitution models (other model parameters) and scoring function as ML inference. In addition, it incorporates prior information about a phylogeny using a prior probability distribution of trees. Felsenstein [63] was the first to suggest the use of Bayesian statistics for reconstructing phylogenies in 1968, but it was not until 1996 that three independent papers [118, 126, 158] showed how posterior probabilities of trees can be calculated under simple priors.

For a number of species s, there exist $T(s)$ alternative tree topologies: τ_1, τ_2, ..., $\tau_{T(s)}$. Given the observed data Δ (the MSA), the posterior probability of observing tree τ_i conditional on the data Δ is calculated with Bayes's theorem as follows:

$$P(\tau_i|\Delta) = \frac{P(\tau_i)P(\Delta|\tau_i)}{\sum_{j=1}^{T(s)} P(\tau_j)P(\Delta|\tau_j)}, \tag{2.13}$$

where $P(\tau_i)$ is the prior probability of tree τ_i, $P(\Delta|\tau_i)$ is the likelihood of tree τ_i, and the denominator is a normalizing constant that involves a summation over all $T(s)$ topologies. The likelihood $P(\Delta|\tau_i)$ is computed by integrating over all possible combinations of branch lengths ϕ and substitution model parameters θ as follows:

$$P(\Delta|\tau_i) = \int_{\phi}\int_{\theta} P(\Delta|\tau_i, \phi, \theta)P(\phi, \theta)d\phi d\theta, \tag{2.14}$$

where $P(\Delta|\tau_i, \phi, \theta)$ can be computed by means of the PLF and $P(\phi, \theta)$ is the prior probability density of the branch lengths and substitution model parameters.

Apart from simple cases, the analytical evaluation of the summation over all $T(s)$ trees and integration over all combinations of branch lengths ϕ and model parameters θ is not possible. For this reason, Markov chain Monte Carlo simulations (MCMC) [79, 86, 129] are employed

to approximate the posterior probabilities of trees. The MCMC approach starts a Markov chain with any combination of a tree topology τ, a set of branch lengths ϕ, and a set of model parameters θ. In each iteration of the Monte Carlo simulation, a change to the state of the chain is proposed. The new state is accepted if it leads to a tree with a better likelihood score. States that lead to worse trees in terms of likelihood score are also accepted with a certain probability. The Markov chain is sampled at given intervals. When the simulation has run long enough, that is, several millions/billions of states have been generated, the fraction of time that the Markov chain spent on a particular state represents a valid approximation of the posterior probability of the corresponding tree. Typically, Metropolis-coupled MCMC [71], a variant of MCMC, is used in phylogenetics to prevent the Markov chain from getting stuck in local optima.

Bayesian phylogenetic inference exhibits advantages and disadvantages when compared to ML. Compute-expensive optimization procedures for instance, such as Newton-Raphson [156] or Brent's algorithm [40], are not required for model parameter and branch length optimization, since both the parameters of the model and the lengths of the branches are sampled via MCMC. On the other hand, there is no confident way to decide how long the simulation should run in order for the Markov chain to converge onto the stationary (equilibrium) distribution. While diagnostic tests can identify MCMC output that has not converged to the stationary distribution [174], convergence to an unknown distribution can not be proven. Consequently, there is no proof that the approximated posterior distribution is accurate enough.

2.5 State-of-the-art software tools

This section describes some state-of-the-art phylogenetic inference tools. Several tools are available online and free of charge. For a comprehensive list you may refer to [7].

Popular phylogenetic packages based on maximum parsimony are PAUP* [194] by David Swofford and TNT [75] by Pablo Goloboff, Steve Farris, and Kevin Nixon. PAUP* (Phylogenetic Analysis Using Parsimony (*and Other Methods)) is a commercial product distributed by Sinauer Associates. It was originally designed as a parsimony-only tool, but support for ML and distance-matrix methods was added later on (version 4.0). It implements an exhaustive tree-search algorithm, although it is probably not used for real-world analyses because of the super-exponential growth in the number of tree topologies (see Section 2.3.2). TNT provides fast tree-search algorithms [74, 140] as well as extensive tree handling capabilities. It is currently the fastest tool for parsimony-based analyses. The tool became freely available a few years ago (2007), but not as open-source code. Recently, Alexandros Stamatakis released what is believed to be the fastest open-source parsimony implementation, *Parsimonator* [177]. It is a light-weight implementation for building parsimony trees. It deploys a randomized stepwise addition order algorithm to build comprehensive trees, and thereafter conducts a few SPR (Subtree Pruning and Regrafting) moves to further improve the parsimony score. It can only accommodate DNA data. The parsimony function implementation has undergone significant performance tuning

using SSE3 and AVX vector intrinsics.

Some of the best-known and most widely used tools for likelihood-based inference are Felsenstein's PHYLIP package [67], PHYML [80], GARLI [4], RAxML [182, 188], PhyloBayes [112], and Mr.Bayes [93, 165]. PHYLIP (PHYLogeny Inference Package) is probably the most widely distributed package for phylogenetic inference. It comprises a collection of 35 open-source tools. The input/output interface of these tools allows to generate tool pipelines to accommodate the requirements of specific analyses. Several inference methods such as distance-matrix methods, maximum parsimony, and maximum likelihood are supported. PHYLIP does not provide performance-optimized implementations but rather sequential proof-of-concept ones. Thus, it is typically not used for real-world analyses but mostly as reference for the development of high-performance tools and the verification of their correctness.

PHYML is developed by Stéphane Guindon and Olivier Gascuel. It is a software tool for ML estimation of phylogenies based on nucleotide or amino acid MSAs. PHYML provides a large number of base substitution models and employs a variety of tree-search strategies. It implements a hill-climbing algorithm that adjusts tree topology and branch lengths simultaneously.

GARLI (Genetic Algorithm for Rapid Likelihood Inference), developed by Derrick Zwickl, is a ML tool that employs a stochastic genetic algorithm to find a "good" tree topology. GARLI searches the tree space with NNI (Nearest Neighbor Interchange) or SPR moves. Furthermore, it can carry out multiple tree searches and bootstrap analyses [66] in a single program run. Bootstrapping [57] is a statistical method for obtaining an estimate of error via dataset resampling. In a phylogenetic context, bootstrapping is used to assess uncertainties in estimated phylogenies.

RAxML (Randomized Axelerated Maximum Likelihood) is developed by Alexandros Stamatakis. It employs heuristics to search the tree space following the approach of a typical hill-climbing algorithm, that is, topological rearrangements such as SPR moves are used to gradually improve the likelihood score of the tree. Furthermore, significant effort has been made to tune performance via technical low-level optimizations such as SSE3 and AVX vectorizations. RAxML exhibits significantly lower memory requirements than other competing codes and typically yields trees with better likelihood scores in less time [182, 188]. Additionally, it is able to perform standard and rapid bootstrap analyses [186]. RAxML is freely available as open-source code. For these reasons, it served as reference for the design and evaluation of the reconfigurable likelihood function architectures presented in Chapter 5.

PhyloBayes is developed by Nicolas Lartillot, Thomas Lepage, and Samuel Blanquart. It is a Bayesian MCMC sampler for phylogenetic reconstruction and molecular dating analyses using nucleotide or protein alignments. The tool is well suited for large multi-gene alignments and employs a large variety of amino acid and nucleotide substitution models.

MrBayes by John Huelsenbeck and Fredrik Ronquist is presumably the most widely used tool for Bayesian phylogenetic inference. It employs a Metropolis-coupled MCMC approach to

estimate posterior probabilities of phylogenetic trees. It is available as open-source sequential and MPI-based implementation [23]. Also, there is a hybrid MPI/OpenMP version [155, 165] by Alexandros Stamatakis and Wayne Pfeiffer.

Chapter 3

FPGA and GPU Architectures

This chapter discusses the hardware architectures of modern FPGAs (Field-Porgrammable Gate Arrays) and GPUs (Graphics Processing Units). Furthermore, the traditional FPGA design flow as well as the vendor-independent OpenCL programming model for GPUs are described.

3.1 Introduction

As general-purpose processors grow in size due to the integration of multi-core CPUs, several concerns have surfaced regarding power consumption and cooling costs. Furthermore, there is an increasing demand for computational power that is driven by the rapid growth in the amount of data to be processed and/or the compute-intensive kernels used. This started paving the way for establishing hardware accelerators such as FPGAs, GPUs, or the Cell Broadband Engine Architecture (CBE) as mainstream HPC alternatives some years ago. Bioinformatics applications like alignment algorithms for instance, have been accelerated with the use of FPGAs [214], GPUs [121], and the CBE [167]. Furthermore, a variety of other scientific or industrial applications (cryptography [215], finance [199], automotive [200], medical imaging [113], aerospace [89], video processing [173], phylogenetics [37]) have witnessed performance boosting due to these hardware (HW) technologies.

FPGAs (Field-Programmable Gate Arrays) are reprogrammable silicon chips. They are field-programmable in the sense that the integrated circuit can be configured after manufacturing, that is, "in the field". A gate array is an integrated circuit dominated by non-interconnected logic gates. Their interconnection is configured *after* fabrication of the chip according to the application at hand, thereby turning the chip into an application-specific integrated circuit (ASIC). The foundational concepts for programmable logic arrays, gates, and logic blocks are credited to David W. Page and LuVerne R. Peterson [151]. In 1985, Ross Freeman and Bernard Vonderschmitt (co-founders of Xilinx [8]) invented the first commercially viable FPGA. Nowadays, FPGAs form the basis of reconfigurable computing. Reconfigurable computing stands in between the high flexibility of microprocessors and the high performance of ASICs by adopting a computation model that deploys hardware to increase performance while retaining much of

the flexibility of a software solution. A comprehensive introduction to reconfigurable computing and the roles of FPGAs in the field can be found in [47] and [87].

GPUs (Graphics Processing Units) are specialized electronic circuits/chips for manipulating computer graphics. GPUs can process large blocks of data in parallel due to their highly parallel architecture. Most modern personal computers are equipped with a GPU, either by means of a dedicated graphics card or on the motherboard. In 1999, NVIDIA introduced the first GPU and provided a technical definition: "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second". A couple of years later, computer scientists started using GPUs in fields other than graphics processing, as for instance medical imaging and electromagnetics, to accelerate programs. Modern GPUs approach parallelism in a processing fashion similar to SIMD computer architectures, that is, by broadcasting the same instruction to multiple execution units. In that sense, GPUs can be regarded as wide SIMD processors. A more detailed introduction to GPU computing can be found in [137] and [138].

The Cell Broadband Engine Architecture (CBE) is the outcome of a joint effort that started in March 2001 by IBM, Sony, and Toshiba to create a power-efficient, cost-effective, and high-performance microprocessor architecture targeting a wide range of applications, including computationally demanding game consoles. The first commercial use of the Cell processor was in Sony's Playstation 3 game console that was released in November 2006. CBE employs Synergistic Processing Elements (SPEs) that handle the computational workload. Each SPE is a RISC processor with 128-bit SIMD units. A Power Processor Element (PPE), based on the IBM PowerPC 64-bit architecture, acts as the controller of the SPEs running conventional operating systems. The on-chip elements (PPEs, SPEs, memory controller, external I/O interfaces) are connected using an internal communication bus, the Element Interconnect Bus (EIB), which is implemented as a circular ring consisting of four 16-byte unidirectional channels. Explicit cache management is required to optimize the use of memory bandwidth in compute- and/or memory-intensive applications. The HW design is based on the analysis of the computational demands of applications such as cyptography, graphics transform and lighting, physics, fast-Fourier transforms (FFT), matrix operations, and scientific workloads. An introduction to and presentation of the Cell architecture can be found in [45] and [97]. The rest of the chapter provides a closer look into the hardware architectures of modern FPGAs and GPUs.

3.2 Field-Programmable Gate Array (FPGA)

There are two leading companies in the field of programmable logic device development, Xilinx [8] and Altera [2]. Both employ different terminologies to describe their FPGA architectures, but the underlying concepts behind the basic building blocks are similar. To avoid confusion, we adopt the Xilinx terminology in this chapter. A white paper by Altera [24] may be used as reference to match the Xilinx terminology with that of Altera.

An FPGA consists of different building components such as input/output blocks or storage elements. At present, several device families are available. All FPGAs in the same device family rely on the same building blocks, but every device contains a different number of them to address the computational demands of different types of applications. Furthermore, the architectural design of the building components usually differs from generation to generation, with newer generations including significant architectural advances and complex designs. This section presents the underlying concepts of the FPGA architecture and describes how a modern FPGA works. Note that the Xilinx Virtex 6 device family [208] is used as reference. Therefore, some topics discussed in this section may only apply to Virtex 6 FPGAs.

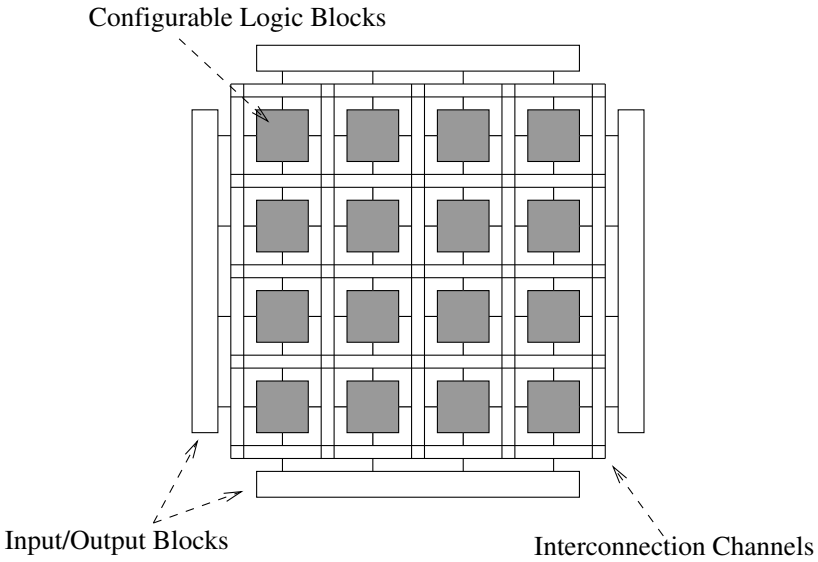


Figure 3.1: Simplified FPGA floorplan.

A typical FPGA architecture can be regarded as a grid of Configurable Logic Blocks (CLBs) with plenty of interconnection resources that is flanked by input/output blocks. Figure 3.1 illustrates a simplified floorplan of a conventional FPGA. The Input/Output Blocks (IOBs) provide the interface between the chip's pins and the internal CLBs. An IOB typically consists of registers, multiplexers, and control as well as clock signals. The CLBs represent the basic logic resources for implementing sequential (the output depends on the input *and* the state of the circuit) and combinatorial (the output depends *only* on the input) circuits. The interconnection channels are used to connect several CLBs with each other in order to implement complex logical functions.

3.2.1 Configurable Logic Block (CLB)

In modern FPGAs, such as the Xilinx Virtex 6 FPGA [208] (henceforth denoted as XV6), a CLB consists of two slices with no direct connections to each other as shown in Figure 3.2. It

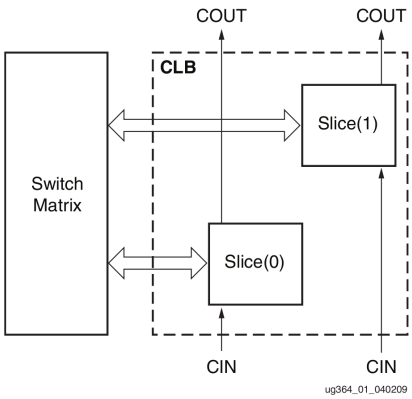


Figure 3.2: Arrangement of slices within the CLB. (Source: [208])

is connected to the routing resources on the chip via a Switch Matrix which allows the signal coming from the CLB to travel out vertically or horizontally. A slice contains logic-function generators (4 in XV6), storage elements (8 in XV6), multiplexers, and carry logic. Two different types of slices are found in a XV6. For low-level details on the architecture of the different slice types see [208]. Figure 3.3 illustrates a superset of elements and connections found in all slices of a XV6. Evidently, a single slice is not very powerful, but when the slice elements are used by all slices, then complex logic and arithmetic operations as well as memory operations can be efficiently implemented.

A logic-function generator is implemented as a look-up table (LUT). A n-input LUT can implement any n-input Boolean function. Figure 3.4 illustrates the equivalence between a simple combinatorial logic circuit and a LUT. The number n of input signals to a LUT varies among FPGA generations and vendors, and typically ranges between 3 bits in older devices and 7 bits in the newer ones. A logic-function generator in a XV6 slice is implemented as a 6-input LUT. Depending on the function implemented, the output of a LUT can either exit the slice, feed the storage elements, or be forwarded the carry-logic chain. As already mentioned, a CLB contains two slices. Each slice can contain one or more logic-function generators. For this reason, a slice is also equipped with a number of multiplexers (3 in XV6) that can be used to combine the output of logic-function generators and implement m-input functions with $m > n$. Employing the multiplexers, a single XV6 slice can implement logic functions with up to 8 input bits. Depending on the configuration of the CLB, LUTs can be combined to form distributed memory blocks for storing larger amounts of data. The storage elements in a slice can be configured as flip-flops or latches with common control signals such as clock, clock enable, and reset. The multiplexers in a CLB can also be combined to design wider multiplexers. The carry-logic chains allow to perform arithmetic addition and subtraction in a slice. A XV6 CLB comprises two separate chains that can also be combined to form wider add/subtract logic.

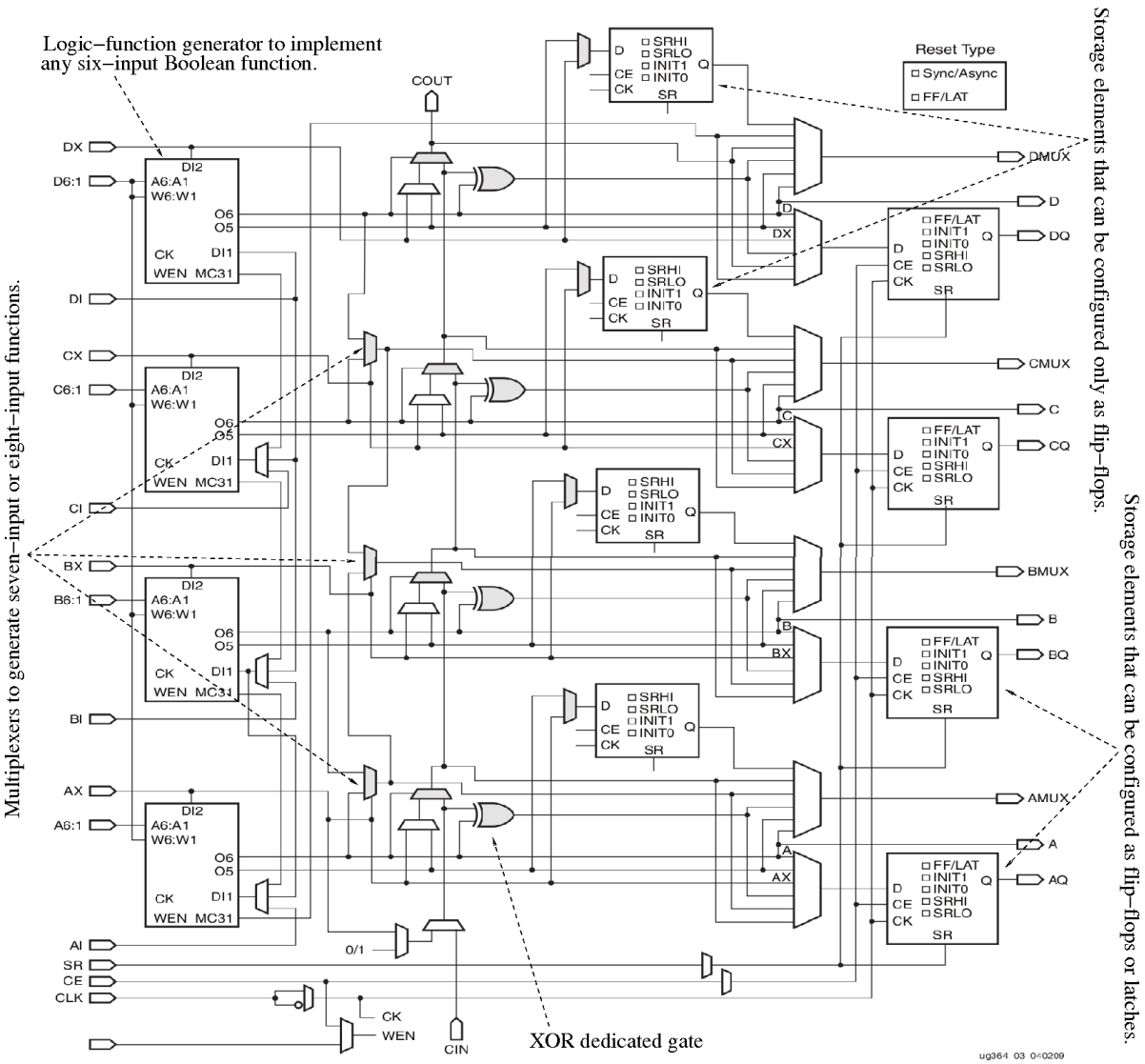
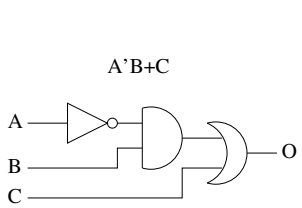


Figure 3.3: Diagram of a Xilinx Virtex 6 slice (SLICEM). (Source: [208])



LUT

A	B	C	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 3.4: Equivalence between combinatorial logic and a look-up table.

3.2.2 Specialized blocks and soft IP cores

Clearly, the CLB slice is the "heart" of a modern FPGA architecture. FPGAs owe the feature of "re-programmability" mainly to the LUT-based logic-function generators that are part of the CLB slices, which, in combination with the flexible routing resources, allow for implementing large and complex designs. Such large circuits usually rely on simple operations or, more accurately, phenomenically simple operations. A seemingly simple operation such as a single multiplication of two operands for instance, can become extremely resource-intensive and complex to implement in digital circuitry. For this reason, FPGAs are equipped with specialized built-in blocks that perform complex but commonly required operations such as floating-point operations for instance. Today's state-of-the-art FPGAs comprise special-purpose blocks for a variety of operations. A quick look into the feature table of the Virtex-6 FPGA family [207] reveals advanced DSP (Digital Signal Processing) slices for arithmetic operations, memory blocks (Block RAM) for fast on-chip data storage, powerful clock managers, interface blocks for PCI Express, MAC (Media Access Controller) blocks for Ethernet communication, and dedicated transceivers.

In addition to the built-in blocks, several soft IP (Intellectual Property) cores are available. Such soft cores, when instantiated in a FPGA design, occupy reconfigurable resources (CLBs) on the chip and hide the complexity of their internal architecture behind a well-defined interface. Evidently, a soft core that performs a specific operation will always deliver worse performance than a hard-coded built-in core for the same purpose, since there will always be a performance overhead induced by the reconfigurability of the hardware that a soft core occupies. Nevertheless, unlike in the case of a built-in core, a soft core does not occupy any chip area when not required by the application design. Soft IP core solutions address a wide range of applications: video controllers, system controllers, memory controllers, communication controllers, arithmetic cores, DSP cores, System-On-Chip components, and processor cores. The most popular online database of IP cores for FPGAs is OpenCores.org (`http://opencores.org/`), which comprises approximately 1,000 IP cores and some 150,000 registered users (April 30^{th}, 2012).

3.2.3 Design flow

A traditional FPGA design flow (Figure 3.5) comprises four stages: i) design entry, ii) synthesis, iii) implementation, and iv) device configuration. Initially, a design (hardware architecture) is described. Design entry approaches can either be schematic-based or HDL-based (Hardware Description Language). Then, the synthesis process generates device-specific netlists (Native Generic Circuit - NGC files in Xilinx terminology) that describe the circuit of the design at the gate level. The synthesis process may generate more than one netlists depending on the complexity and size of the design. Thereafter, the implementation process begins. It is a sequence of three steps: i) Translate, ii) Map, and iii) Place & Route. During the first step, the generated netlists and design constraints are combined into a logic design file (Native Generic

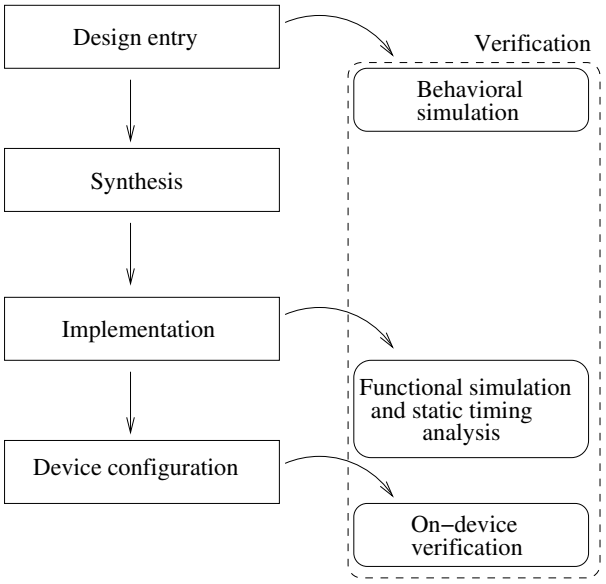


Figure 3.5: Traditional FPGA design flow.

Database - NGB). Design constraints are provided in the form of a text file (User Constraints File - UCF) which specifies time requirements and assigns physical elements such as pins or switches to the top-level ports of the design. The mapping process divides the circuit of logic elements into blocks that fit into the physical building blocks of the FPGA, such as CLBs, IOBs, and others (see Section 3.2.2). The outcome of the mapping process is a NCD file (Native Circuit Description). The last step of the implementation process is to place the blocks from the mapping process into logic blocks in accordance with the constraints (UCF), and to define how the routing resources will be used to connect the blocks. After Place & Route, a completely routed NCD file is generated. This file is converted to an FPGA-friendly format (bitstream) and downloaded to the FPGA via a cable.

Several verification attempts can take place at intermediate steps of the design process (Figure 3.5). Behavioral simulation, the first in the series of verification steps, takes place before synthesis by employing a high level of abstraction to model a HDL-described design. It verifies the HDL syntax and confirms that the design is functioning as specified. Behavioral simulations identify errors early in the design process, thereby allowing the designer to fix them inexpensively/quickly as opposed to on-device debugging. After the synthesis process, functional simulations on a purely structural description of the design can be used to verify functionality. A functional/structural simulation takes longer than a behavioral simulation, but it provides a more in-depth analysis of the design. A static timing analysis (timing simulation) is carried out when the implementation step is completed. It calculates the worst-case place-and-route delays for the signals of the design and verifies the operation of the circuit. A timing simulation takes even longer than both behavioral and functional simulations, but it also provides more

details on the implementation of the design on actual hardware. Finally, when the bitstream is downloaded to the chip and the FPGA is configured, on-device testing can be deployed to verify the correctness of the design. Errors that were not detected during the simulation steps may occur when the design is implemented on actual hardware. On-chip verification requires a specific download cable and software that allow for monitoring pre-defined (during the design entry stage) data buses for a short period of time (some hundreds of clock cycles). Throughout the design and verification process, every change in the design requires repeating all these steps from scratch. Therefore, on-chip verification is the most time-consuming and difficult verification step, but also the one that guarantees that the hardware behaves as specified during the design entry stage.

3.2.4 HDL-based design entry

At present, the most popular hardware description languages are Verilog and VHDL. Verilog was introduced in 1985 by Gateway Design Automation which was acquired by Cadense Design Systems [3] in 1989. VHDL, which stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, was developed in 1987 by the U.S. Department of Defense. The very first use of Verilog and VHDL was to simulate circuit designs that were already described at the schematic level. HDLs enable engineers to work at a higher level of abstraction than the schematic level. Although schematic-described designs almost always outperform HDL-described equivalents, the increase in productivity has established HDLs as the natural choice for digital design, whereas schematic-based approaches are nowadays only deployed for designs with extreme requirements with respect to performance, resources, or power consumption. In the following, the VHDL language is briefly described.

VHDL is based on the Ada programming language. Therefore, it is strongly typed and not case-sensitive, but exhibits an extended set of Boolean operators that are commonly used in hardware. Unlike procedural languages such as C, VHDL is a dataflow language that describes concurrent systems. A design process that starts with a VHDL code ends with the configuration of actual hardware rather than the "execution" of the VHDL code. A design described by a VHDL file consists of an 'entity' part and an 'architecture' part. Complex designs may consist of many VHDL files. One may consider every single file as an independent module of the architecture. The 'entity' part is the interface of the module, whereas the 'architecture' part is the logic circuit. An HDL-based design typically begins with a high-level schematic-like architectural diagram. This is mostly an abstract representation of the architecture's modules and the connections between them. Thereafter, the process of creating the hardware description (writing HDL code) highly depends on the nature of the circuit and the engineer's programming style. Eventually, the architecture's modules are implemented in one or more VHDL files and connected together in a top-level VHDL file. An example describing a 64-bit 2-to-1 multiplexer in VHDL is shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX2to1_64 is -- Entity's name
    Port ( sel : in  STD_LOGIC; -- 1-bit input signal
           A : in  STD_LOGIC_VECTOR (63 downto 0); -- 64-bit input signal
           B : in  STD_LOGIC_VECTOR (63 downto 0);
           RES : out  STD_LOGIC_VECTOR (63 downto 0)); -- 64-bit output signal
end MUX2to1_64;

architecture Structural of MUX2to1_64 is

-- Declaration of internal 64-bit signals
signal sel_vec, n_sel_vec, t_res: std_logic_vector(63 downto 0);

begin

-- 1-bit signal 'sel' is connected to all 64 bits of 'sel_vec'
sel_vec<=(others=>sel);

-- Inverted 1-bit signal 'sel' is connected to all 64 bits of 'n_sel_vec'
n_sel_vec<=(others=>not sel);

-- The results of two 64-bit wide 'and' operations are wired to
-- a 64-bit wide 'or' gate.
t_res<= (n_sel_vec and A) or (sel_vec and B);

-- The output of the 'or' gate is connected to the entity's output signal 'RES'
RES<=t_res;

end Structural;
```

3.3 Graphics Processing Unit (GPU)

Initially, Graphics Processing Units were built to accommodate the computational demands of the graphics pipeline, a computer graphics term that refers to the state-of-the-art method of rasterization-based rendering or simply, the conversion of three-dimensional scenes into two-dimensional images for output on a display. Today, GPUs exhibit a highly parallel HW architecture with enormous arithmetic capabilities. As such, they are increasingly employed for the acceleration of a variety of computationally intensive scientific applications. Unlike typical

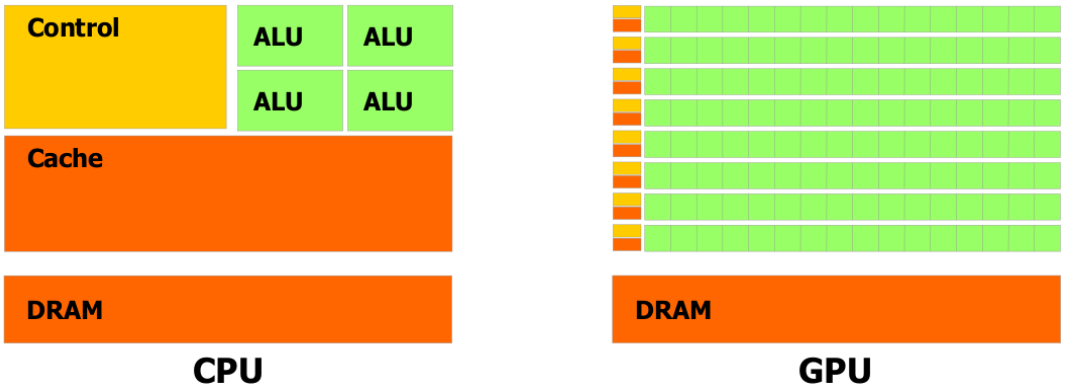


Figure 3.6: GPUs devote more transistors to data processing. (Source: [143])

CPU architectures, GPUs devote more transistors to data processing rather than data caching and control flow (see Figure 3.6).

3.3.1 Architecture of a GPU

Currently, two companies are dominating the GPU market, NVIDIA [5] and AMD [1]. They both build architectures with unified and massively parallel programmable units [149]. An excellent general introduction to GPU computing can be found in [149]. To provide a closer look into GPU architectures, we describe the NVIDIA general-purpose parallel computing architecture named CUDA [143] (Compute Unified Device Architecture) and the Fermi architecture [144] as an example.

CUDA was introduced in November 2006. It comes with a parallel programming model that enables programmers to efficiently develop code for complex computational problems. According to the CUDA architecture, a GPU is built around an array of so-called Streaming Multiprocessors (SM). SM performance benefits from i) Instruction Level Parallelism (ILP) within a single thread and ii) the fact that hundreds of threads can be executed concurrently. To accommodate this immense number of threads, a SIMT (Single Instruction, Multiple Threads) computing model is employed. A multi-threaded program is organized in blocks of threads that execute independently of each other on the SMs. This approach allows GPUs with more SMs to automatically execute the code faster (see Figure 3.7).

CUDA is a heterogeneous programming model that assumes a *host* C code running on a CPU and a co-processor GPU *device*. The compute-intensive parts of a CUDA application are executed on the GPU device by launching parallel kernels. A parallel kernel executes across a set of threads which are organized in thread blocks. The thread blocks are organized in grids of thread blocks. Every thread block executes on a SM which consists of 32 CUDA cores. A CUDA core (Figure 3.8) comprises a fully pipelined integer arithmetic and logic unit (ALU) as

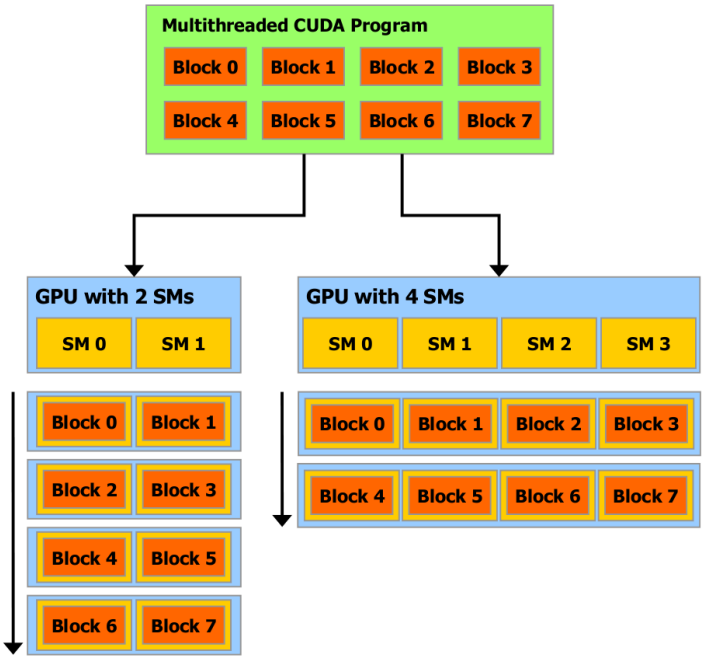


Figure 3.7: A GPU with more SMs (Streaming Multiprocessors) automatically executes the load of thread blocks faster. (Source: [143])

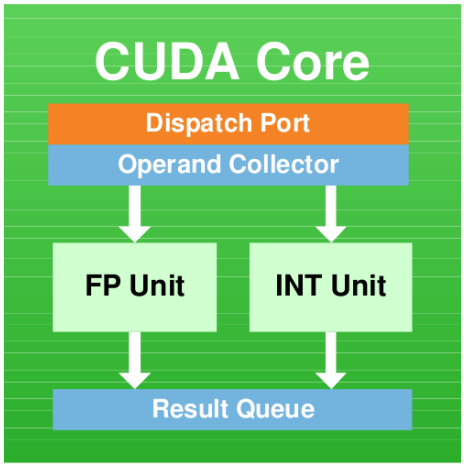


Figure 3.8: Schematic representation of a CUDA core. (Source: [144])

well as a floating-point unit. Every thread executes sequentially on a single CUDA core.

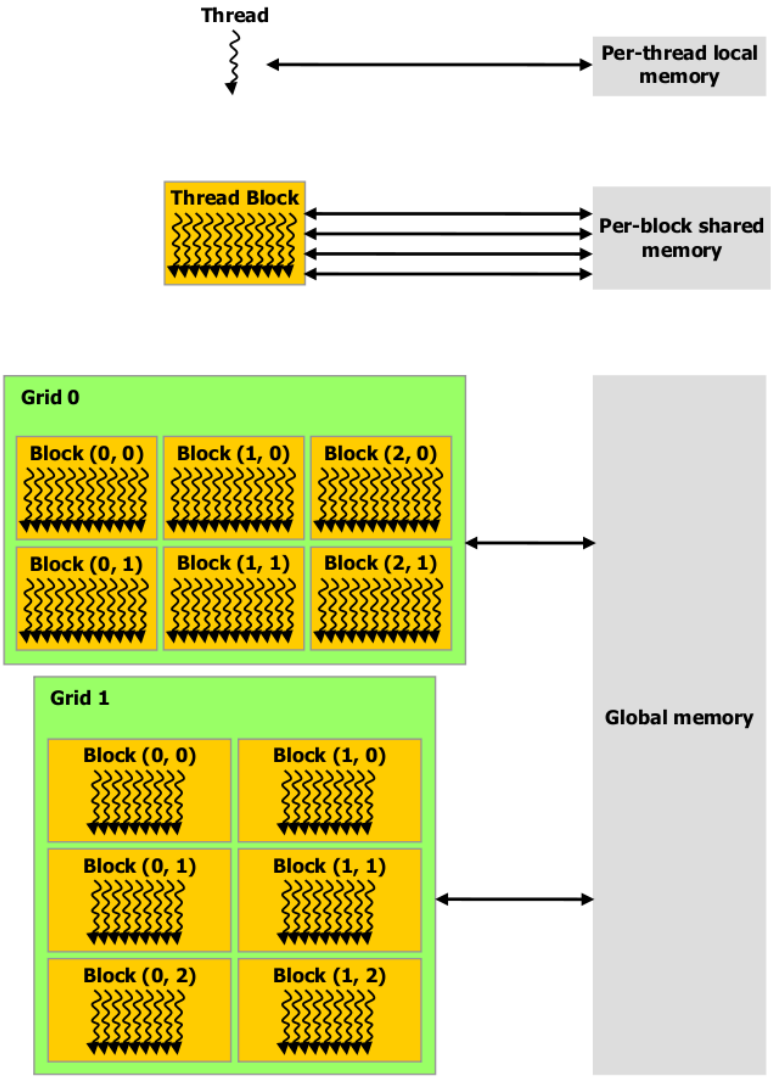


Figure 3.9: Memory hierarchy in the CUDA programming model. (Source: [143])

The CUDA programming model assumes separate memory spaces for the *host* and the *device*. Furthermore, a CUDA thread has access to several on-device (GPU) memory spaces such as private local memory, shared memory, and global memory. The private memory space is visible to each thread exclusively and usually used for storing registers, function calls, and array variables. The shared memory space is available per-(thread)block and facilitates data sharing among threads within a thread block. Data sharing among thread blocks is implemented via the global memory space. Figure 3.9 illustrates the memory hierarchy in the CUDA programming model. The current CUDA generation (Fermi architecture) is shown in Figure 3.10. For an

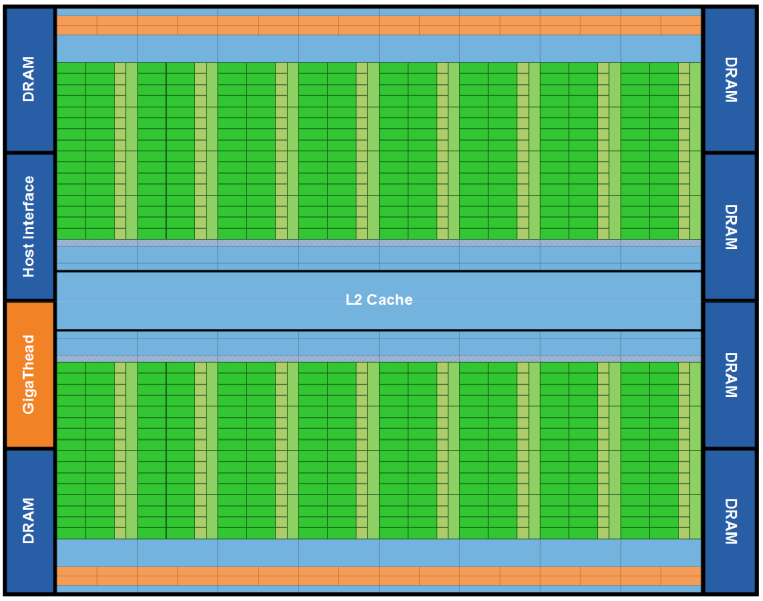


Figure 3.10: Overview of the Fermi architecture. Fermi's 16 SMs are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contains an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache). (Source: [144])

in-depth description of the Fermi architecture refer to [144].

3.3.2 OpenCL programming model

OpenCL (Open Computing Language [104]) is an open standard for parallel programming of heterogeneous systems. An OpenCL application typically runs on a host CPU and one or more GPUs. The OpenCL architecture is similar to CUDA.

A CUDA device consists of several *Streaming Multiprocessors* (SMs) which correspond to the OpenCL *compute unit* terminology. The OpenCL *work-items* and *work-groups* correspond to the CUDA *thread* and *thread block* concepts. In analogy to CUDA applications, OpenCL applications consist of a *host* program which is executed on a CPU and one or more *kernel* functions that are offloaded to the GPU(s). The kernel code executes work-items/threads sequentially and work-groups/thread blocks in parallel. Since threads are organized into thread blocks, thread blocks are in turn organized into *grids of thread blocks*. An entire kernel is executed by such a grid of thread blocks.

OpenCL applications can access various types of memory: global, local, shared, constant, and texture. The global memory resides in the device memory (e.g., DRAM on the GPU board) and is accessed via 32-, 64-, or 128-byte transactions. Global memory is allocated on a per-application basis and can be accessed by all work-items and work-groups. To maximize global

memory throughput, it is essential to maximize memory coalescence and minimize address scatter.

CUDA local memory accesses typically occur for automatic variables that the compiler generates and places in local memory (e.g., large structures or variables that do not fit into registers). Since CUDA local memory resides in device memory, local memory accesses exhibit a similarly high latency and low bandwidth as global memory accesses. On the other hand, shared memory resides on-chip and is, therefore, substantially faster than local and/or global memory. In OpenCL, local memory is located in shared memory and can be accessed on a per-thread basis. Shared memory can be accessed on a per-(thread)block basis. Thus, as long as there are no bank conflicts, using shared memory allows for attaining high memory bandwidth. An L1 cache for each SM and an L2 cache shared by all SMs are used to cache accesses to local or global memory.

Finally, constant and texture memories reside in device memory. The constant memory is a read-only, specialized region of memory that is suitable for broadcast operations. Each SM has a read-only uniform cache and a read-only texture cache to speed up reads from the constant and the texture memory spaces, respectively. The texture cache is suitable for dealing with two-dimensional textures because it is optimized for two-dimensional spatial locality.

Chapter 4

Parsimony Reconfigurable System

This chapter presents a reconfigurable architecture for computing parsimony scores on trees of realistic size. Furthermore, a fully functional CPU-FPGA prototype system is described.

4.1 Introduction

Parsimony strives to find the phylogenetic tree that explains the evolutionary history of organisms by the least number of mutations. The phylogenetic parsimony function is a popular, discrete criterion for reconstructing evolutionary trees based on molecular sequence data. Compared to the likelihood criterion, parsimony requires significantly less memory and computations to calculate the parsimony score of a given tree topology, which is important for analyzing very large datasets [76]. Because parsimony is a discrete function, it fits well to FPGAs. This chapter presents a versatile FPGA implementation of the parsimony kernel and compares its performance to a highly optimized SSE3- and AVX-vectorized software implementation. The hardware description of the reconfigurable architecture is available as open-source code at: `http://www.exelixis-lab.org/FPGAMaxPars.tar.bz2`.

Kasap and Benkrid [100, 101] recently presented the—to the best of our knowledge—first reconfigurable architecture for the parsimony kernel and assessed performance on a FPGA supercomputer by exploiting fine-grain and coarse-grain parallelism. The implementation is limited to trees with a maximum size of 12 organisms, which are very small by today's standards; the largest published parsimony-based tree has 73,060 taxa [76]. To find the tree with the globally best parsimony score, the authors use an exhaustive search algorithm and evaluate all possible trees with 12 organisms in parallel. An evaluation of all possible trees, even in parallel, is evidently not possible for parsimony-based analyses of larger trees because of the super-exponential increase in the possible number of trees (see Section 2.3.2). Parsimony-based programs for large datasets deploy heuristic search strategies, e.g., subtree pruning and regrafting (SPR [195]) or tree bisection and reconnection (TBR [195]). These search strategies (as implemented for instance in TNT, `Parsimonator`, or PAUP*) do not require a *de novo* computation of the parsimony score based on a full post-order tree traversal as implemented

in [100, 101]. Instead, they only require the update of a comparatively small fraction of ancestral parsimony vectors. Hence, a fundamentally different approach to implement the parsimony function on a reconfigurable architecture for such commonly used heuristic search strategies is required.

Kasap and Benkrid report speedups between a factor of 5 and up to a factor of 32,414 for utilizing 1, 2, 4, and 8 nodes (each node is equipped with a Xilinx Virtex4 FX100 FPGA) on the Maxwell system [31] compared to a 2.2 GHz Intel Centrino Duo processor. However, the speedups reported are only relative speedups with respect to the parsimony implementation in PAUP* [194] and not with respect to the fastest implementation of parsimony in the TNT program [75]. Note that neither PAUP* nor TNT are open-source and, therefore, an accurate performance analysis/comparison is not possible. For this reason, we use the open-source code `Parsimonator` [177], which implements a representative search strategy based on SPR moves. The parsimony kernel in `Parsimonator` is highly optimized and the program has been used to compute parsimony trees on DNA datasets with up to 116,408 organisms and ten genes.

The remainder of this chapter is organized as follows: Section 4.2 describes the parsimony kernel, Section 4.3 outlines the reconfigurable architecture, and Section 4.4 presents implementation details and performance results.

4.2 Parsimony kernel

The parsimony kernel operates directly on the MSA and the tree. The sequences in the MSA are assigned to the leaves of the tree and an overall score for the tree is computed via a post-order tree traversal with respect to a virtual root. An important property of the parsimony function is that parsimony scores are invariant to the placement of such a virtual root. Parsimony is characterized by two additional properties: i) it assumes that MSA columns have evolved independently, that is, given a fixed tree topology, one can simultaneously compute the parsimony score for each MSA column in parallel. To obtain the overall score of the tree, the sum over all m per-column parsimony scores at the virtual root is computed. ii) parsimony scores are computed via a post-order tree traversal that proceeds from the tips toward the virtual root and computes ancestral parsimony vectors of length m at each inner node that is visited (see Figure 4.1 and Felsenstein's pruning algorirthm in Section 2.4.2.3). The ancestral parsimony vectors are calculated using Sankoff's algorithm [172].

The parsimony criterion intends to minimize the number of nucleotide changes on a tree. Hence, for a given/fixed tree topology, we need to compute the smallest number of changes (mutations) required to generate the tree. This minimum number can be computed via a post-order traversal of the tree. Given an arbitrarily rooted tree, one can proceed bottom up from the tips toward the virtual root to compute ancestral parsimony vectors and count mutations based on the two previously computed child vectors of the post-order traversal. To store tip vectors (containing the actual DNA data) and ancestral parsimony vectors (containing the ancestral

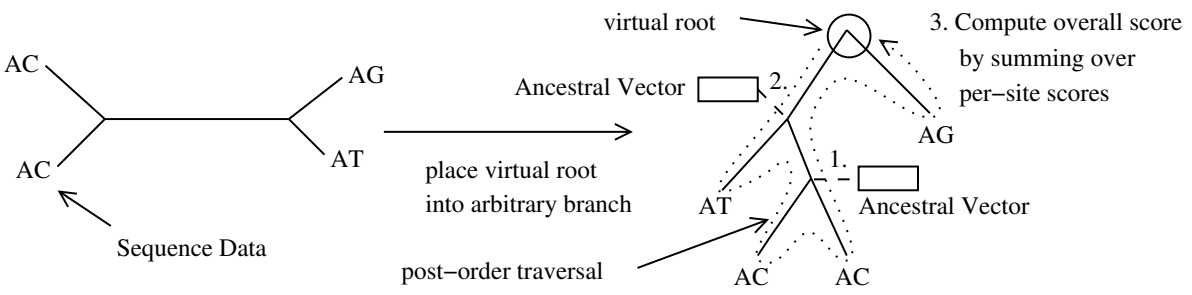


Figure 4.1: Virtual rooting and post-order traversal of a phylogenetic tree.

sequences), we need to allocate 4 bits per alignment site i at each node of the tree. Hence, the total memory required to store the parsimony vectors is $m \cdot 4 \cdot (2n-2)$ bits, where m is the number of sites and $2n-2$ the total number of inner and outer nodes in an *unrooted* binary tree (n is the number of tips). In the following we will only describe the computational steps required to compute the parsimony score on a tree (see [172] for a justification and further details).

The parsimony vectors (bit vectors) at the tips are initialized as follows: for a nucleotide `A` at a position i, where $i = 0...m-1$, we assign `A:=1000` (respectively `C:=0100`, `G:=0010`, `T:= 0001`). When the tip vectors have been initialized, one can start computing the parsimony score of the tree. We will focus on computing the parsimony score ps_i (minimum number of mutations) for a single site i, since the overall score is simply the sum over all per-site scores at the virtual root: $\sum_{i=0}^{m-1} ps_i$.

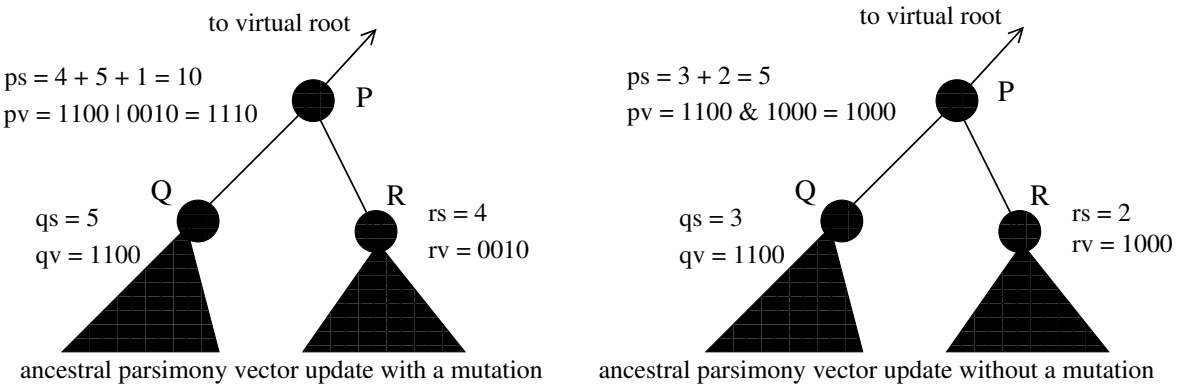


Figure 4.2: Parsimony vector and score updates with and without mutation.

Given two already computed child vectors q and r, we compute the parent vector p at site i as follows (see Figure 4.2). The parsimony score is initially set to the sum of the parsimony scores of two child vectors $ps_i := qs_i + rs_i$, that is, we take into account how many mutations were required to explain the two subtrees rooted at q and r for site i. Then, we compare the 4-bit vectors of q and r with a bit-wise `and` operation. If this bit-wise `and` yields 0, this means, for instance, that site i in subtree q only contains `A`s ($qv_i = 1000$) and r only contains

Cs ($rv_i = 0100$) at site i. Hence, we need to add a mutation and increment the parsimony score by one $ps_i := ps_i + 1$. The parsimony vector at position i of p is then calculated as $pv_i := qv_i$ $OR_{bit-wise}$ rv_i, that is, we conduct a bit-wise `or` on qv_i and rv_i to obtain a new state that now comprises A or C. Thus, $pv_i := 1100$, which means that the ancestral state can be A or C because we have already counted the required mutation. If the initial bit-wise `and` on qv_i and rv_i does not yield zero, we do not need to count an additional mutation. In this case, we simply set $pv_i := qv_i$ $AND_{bit-wise}$ rv_i, thereby saving the shared state between qv_i and rv_i in pv_i. When the virtual root is reached, we conduct exactly the same computations on child vectors q and r to update the parsimony score at the root p, but we do not store the ancestral state pv since we are only interested in the score (mutation count) at p.

4.3 Reconfigurable architecture

In the following we describe the reconfigurable architecture. We denote ancestral vector computations as *NV* operations and score computations at the virtual root by *EV*.

4.3.1 Processing unit

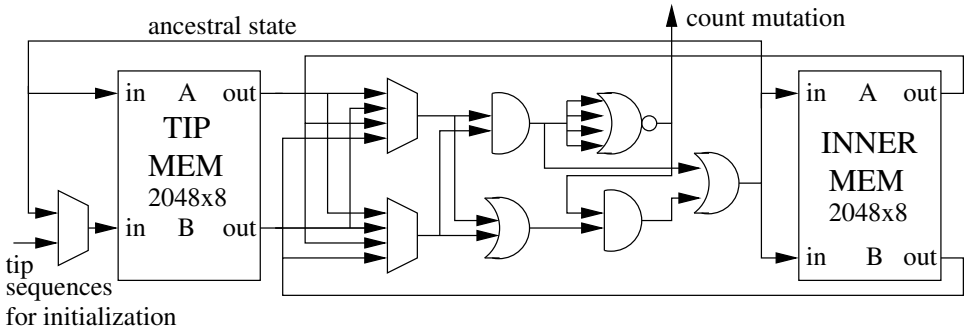


Figure 4.3: Architecture of the basic parsimony processing unit.

Figure 4.3 illustrates the basic processing unit (PRU) of the reconfigurable parsimony kernel. Each PRU operates on two child vector entries (two sites). The PRU architecture deploys a pair of dual-port memories, i.e., one memory instance is used for storing tip vectors and one for storing inner vectors. Each memory instance can store a maximum of 2,048 addressable bytes. The rationale for selecting this specific memory size is that thereby we only occupy a single 18-Kb Block RAM slice per PRU memory instance. If each PRU requires only a limited amount of memory blocks, the overall reconfigurable parsimony system can be extended by additional PRUs in a seamless way (see Section 4.3.2).

To initiate a parsimony analysis, only the TIP MEMORY has to be initialized with the bit-encoded DNA sequences in the MSA. Every tip and inner vector is assigned a static address space in the respective memory prior to executing any operation. During a post-order tree

traversal, the following three memory access cases can occur regarding the child input vectors at nodes q and r: i) q and r are both tips, ii) either q is a tip or r is a tip, and iii) q and r are inner nodes. For the TIP-TIP *and* TIP-INNER cases, the input vectors are retrieved and read from the corresponding TIP and INNER memories, respectively. The result vector at node p (when it needs to be stored for a NV operation) is stored in the INNER memory. In analogy, a NV operation for the INNER-INNER case would require an INNER memory with three memory ports: two ports for reading the q and r vectors and a third port for writing the p vector. To efficiently implement the INNER-INNER case for NV (where p, q, and r are inner nodes) using present FPGA technology that only provides two ports per memory block, the p vector is temporarily stored in a special memory. We denote this special memory, which forms part of the TIP MEMORY, as EXTRA space. At each clock cycle, two 4-to-1 multiplexers (see Figure 4.3) are used to select the correct memory buses containing valid vector input data. The multiplexer selection bits denote the corresponding TIP-TIP, TIP-INNER, and INNER-INNER cases. The group of logic gates in the center of Figure 4.3 implements the bit-wise operations to compute the parsimony kernel (see Section 4.2). An additional 2-to-1 multiplexer is used to distinguish between the two data buses that provide input to the second TIP MEMORY port. One bus provides the DNA sequences of the MSA during the memory initialization process while the second bus provides ancestral states during NV and EV operations if the EXTRA space of the TIP MEMORY is used.

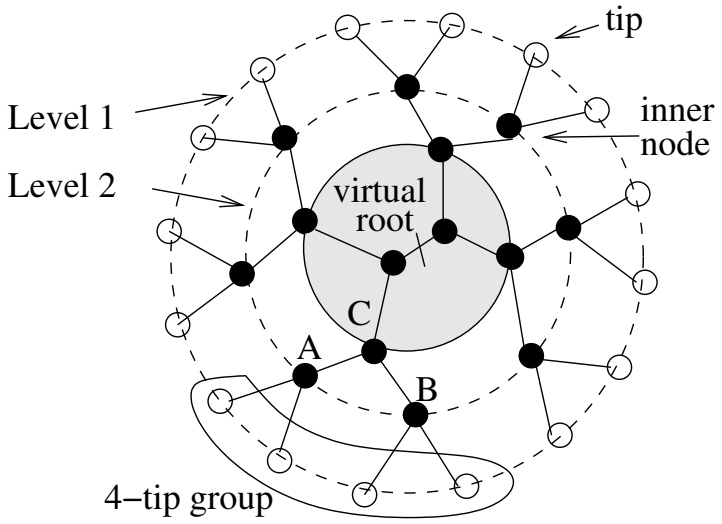


Figure 4.4: Worst-case tree topology in terms of EXTRA space requirements.

The size of the EXTRA space depends on the dimension of the input dataset, i.e., the number n of taxa in the MSA and the number m of nucleotides per DNA sequence. It also depends on the tree shape. Figure 4.4 illustrates the worst-case tree in terms of EXTRA space requirements. Fully balanced trees require maximum EXTRA space, which amounts to 50% of the memory required to store the input tip sequences in TIP MEMORY. In a fully balanced tree,

for every group of four tips, one inner node needs to be stored in EXTRA space. In Figure 4.4, the highlighted group of four tips at level 1 (tip level) has two parent nodes/vectors at level 2 (one level closer to the virtual root). Vectors A and B (the direct ancestors of the tips), can be stored in INNER MEMORY. Since both A and B are inner vectors (stored in INNER MEMORY), their common ancestor C must be stored in EXTRA space to avoid a memory port conflict. In this worst-case scenario, every inner vector in the highlighted gray area of Figure 4.4 must be stored in EXTRA space. Decisions for writing inner vectors to EXTRA space are orchestrated by an appropriately adapted `Parsimonator` version. Thus, a dedicated, reconfigurable EXTRA space control unit is not necessary. For a fully balanced tree with n tips, the maximum number of inner nodes IN_EX that need to be stored in EXTRA space during a phylogenetic analysis is given by the following equation:

$$IN\_EX = \begin{cases} n/2 - 2 & \text{if } n \mod 4 = 0 \\ (n + 4 - n \mod 4)/2 - 2 & \text{if } n \mod 4 \neq 0. \end{cases}$$

4.3.2 Pipelined datapath

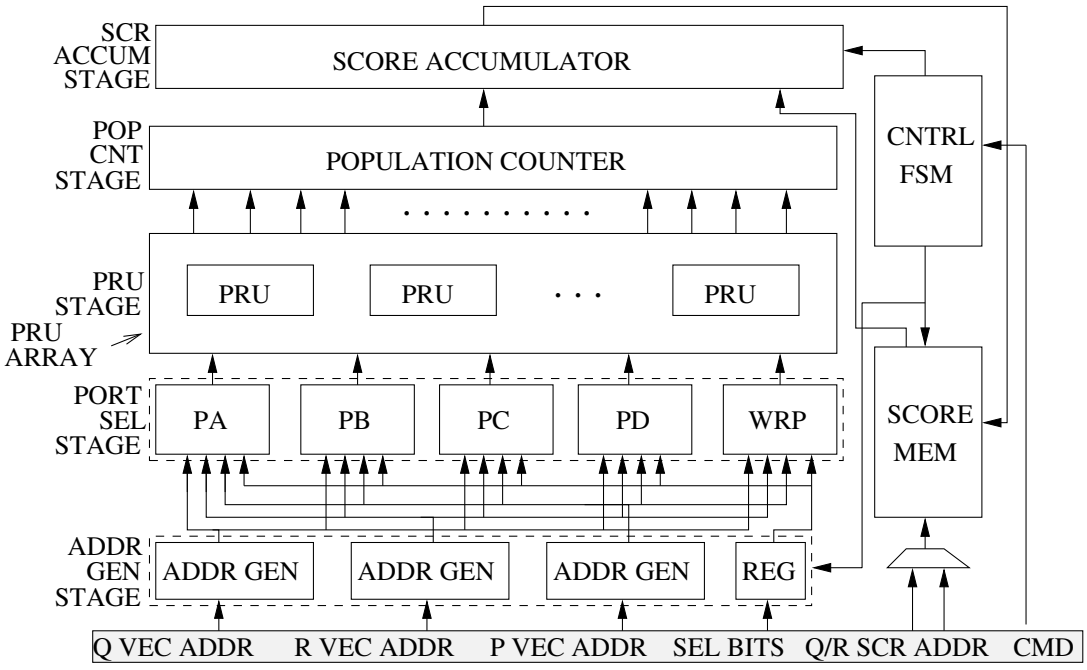


Figure 4.5: Top-level design of the pipelined architecture.

The generic input command that must be issued to the pipeline during a clock cycle to initiate parsimony computations is highlighted at the bottom of Figure 4.5. Both operations (NV and EV) require a set of four 2-byte read addresses; they contain the start addresses of the parsimony child vectors (*Q VEC ADDR*, *R VEC ADDR*) and corresponding parsimony scores

(*Q SCR ADDR*, *R SCR ADDR*) of child nodes q and r. A *NV* operation also requires two additional write addresses for storing the parent vector p and the respective score at p.

The top-level design of the pipelined datapath has five stages (see Figure 4.5). Read/write memory addresses are initially generated by using 11-bit counters during the address generation stage (ADDR GEN). In the next pipeline stage (PORT SEL), the PA, PB, PC, and PD components implement multiplexers and logic that decides to which three (out of four) PRU memory ports the read/write addresses will be sent. The WRP component generates the write enable signal for the selected write port.

The PRU array comprises several parallel and completely independent PRUs. The number of PRUs determines the array size. All PRUs in the array receive the same read/write addresses, write enable signal, and selection bits to perform the same operation on different parsimony vector entries (sites of the MSA). Each PRU contains two memory components and logic. Since the PRU array is a vector-like component (each PRU operates independently), using only two 18-Kb Block RAM slices allows for tailoring the array size to the available FPGA according to the number of available memory blocks on the device. Therefore, we implemented a program called *FPGA_MaxPars_Gen* (included in FPGA_MaxPars.tar.bz2) for generating VHDL wrapper files. Thereby, one can instantiate PRU arrays with user-defined length/number of PRUs.

The POP CNT pipeline stage contains a population counter (to count the number of bits set to 1) for the computation of partial parsimony scores across alignment sites (PRUs). If one regards the tip and inner memory instances of the PRU array as two larger memory components, each with depth of 2,048 and width that depends on the array size (number of PRUs), then a partial score refers to the total number of mutations across sites that can be stored in a PRU array memory line (see Section 4.3.3 for details on the population counter).

Finally, in the SCR ACCUM stage, the parsimony score is computed with three 32-bit adders. The SCORE MEM memory is used to store intermediate scores (parsimony scores for each inner node that defines a subtree). One adder is used to calculate the sum of the input child node scores at q and r. The scores for q and r are retrieved from the SCORE MEM based on the input SCR ADDR addresses. The second adder/accumulator sums up the partial scores that are produced by the population counter. The last adder computes the final score by adding the output of the accumulator to the sum of the scores at q and r. For *NV* operations, the parsimony score is written to SCORE MEM.

The size of SCORE MEM (2,048 integers) is the upper limit for the number n of organisms (DNA sequences) the architecture can accommodate. This maximum number of 2,048 organisms decreases in proportion to the number of PRU array memory lines required by each sequence. The number of required PRU array memory lines increases with the number m of MSA sites/columns since, for long MSAs, each organism will occupy more than one PRU array memory lines.

4.3.3 Population counter

The population counter is implemented as a tree of adders with increasing width, i.e., at each level of the adder-tree the adders are one bit wider than at the previous level. The input bit vector size for the first level depends on the PRU array size. The stand-alone FPGA_MaxPars_Gen software can be used to generate a population counter with user-defined size and latency. Pipeline registers are optionally inserted as needed between adder levels in the tree to alleviate the negative impact of a very large (deep) population counter on the overall operating clock frequency. To the best of our knowledge, FPGA_MaxPars_Gen is the only open-source population counter generator for FPGAs. Because the latency of the population counter influences the total latency of the parsimony architecture pipeline, FPGA_MaxPars_Gen instantiates a shift register (in the VHDL wrapper file) to synchronize the population count computations with the rest of the system.

4.4 Implementation

We describe the verification procedure for the reconfigurable architecture in Section 4.4.1. Then, we present a PC-FPGA prototype system (Section 4.4.2) and a performance evaluation for a larger reconfigurable system (Section 4.4.3).

4.4.1 Verification

Initially, we modeled our architecture (including the address assignment to EXTRA space) in `Parsimonator` using the C programming language. We replaced the standard *NV* and *EV* functions (accounting for 99% of total execution time) by implementations that reflected the reconfigurable architecture to assess and verify the correctness of the approach.

Thereafter, the reconfigurable architecture was implemented in VHDL and mapped on a Virtex 5 SX95T-1 FPGA. We verified the correctness of the hardware system by extensive post-place-and-route simulations using Modelsim 6.3f by Mentor Graphics and tests on an actual chip using a HTG-V5-PCIE development board with a Virtex 5 SX95T FPGA. Chipscope Pro Analyzer was used to monitor the input and output ports of the design.

4.4.2 Prototype system

After successful verification, a fully operational PC-FPGA prototype system was designed to test this implementation of `Parsimonator` using real biological datasets on an actual board. We used the C interface of our open-source PC $\leftrightarrow$ FPGA communication platform [10] (available at `http://opencores.org/project,pc_fpga_com`) to transfer bit-encoded DNA sequences and issue NV/EV commands to the board. On the FPGA side, the DNA sequences were used to initialize the TIP MEMORY, and the NV/EV commands were used to trigger computations. The receiving background reader mechanism provided by this platform was used to receive

parsimony scores on the PC side after an EV command had been issued to the board. The FPGA_MaxPars_Gen program was used to generate a PRU array of size/width 64, as well as a correctly sized population counter for the prototype system, that is, 64 PRUs were instantiated on the device allowing for 128 alignment sites to be processed simultaneously (remember that each PRU can compute the parsimony score for two sites; see Section 4.3.1).

4.4.3 Performance

To present a fair performance assessment for our accelerator architecture, we created a high performance instance of the architecture (using FPGA_MaxPars_Gen) with an array of 512 PRUs and mapped it on a Virtex 6 SX475T-2 FPGA. Furthermore, we vectorized `Parsimonator` with 256-bit AVX SIMD instructions. An evaluation of the prototype and the high-performance systems regarding resources and clock frequencies is provided in Table 4.1. Note that the currently largest available FPGA with respect to available Block RAM slices (Virtex 7 VX865T) can accommodate an array of 1,800 PRUs, which allows for computing 3,600 sites in parallel. Table 4.2 shows execution times (in seconds) for real-world biological datasets using the SSE3

	64-PRU System	512-PRU System
Device	Virtex 5 SX95T	Virtex 6 SX475T
Slice Registers	5,568(9%)	41,091(6%)
Slice LUTs	4,133(7%)	22,520(7%)
Occupied Slices	1,933(13%)	9,608(12%)
Block Rams (18-Kb)	132(27%)	1,028(48%)
Frequency (MHz)	192.374	188.456

Table 4.1: Resources/performance of the Virtex 5 and the Virtex 6 systems.

and AVX versions of `Parsimonator` (using one core of an Intel i7 2600 CPU running at 3.40 GHz), and the reconfigurable architecture with 512 PRUs (mapped on the Virtex 6 device). The FPGA accelerator is up to 9.65 times faster than the optimized software in the best case.

Dataset	Execution times			Speedup FPGA vs	
taxa-sites	SSE3	AVX	FPGA	SSE3	AVX
100-48965	1.48	0.91	0.15	10.12	6.22
125-16503	1.59	0.92	0.16	9.69	5.6
150-871	0.13	0.10	0.01	12.55	9.65
218-1318	0.40	0.26	0.04	9.08	5.9
354-224	0.24	0.21	0.04	6.41	5.61
500-759	0.74	0.52	0.06	11.56	8.12

Table 4.2: Execution times (in seconds) for NV/EV invocations.

4.5 Summary

This chapter presented a reconfigurable architecture for computing the parsimony function on evolutionary trees of realistic size. The architecture is adapted to the computational requirements of modern tree-search strategies and has been demonstrated to work in a fully functional PC-FPGA setup, where the PC steers the computations on the board by using our PC-FPGA communication library.

Previous work on the acceleration of the parsimony kernel using FPGAs reported speedups between a factor of 5 and up to a factor of 32,414 for the platforms that were used (Maxwell system with Virtex 4 FPGAs versus Intel Centrino Duo at 2.2 GHz). However, this chapter revealed that, when the reconfigurable system is designed to meet real-world requirements such as being able to accommodate heuristic tree-search strategies and trees of realistic size, although FPGAs can still outperform CPUs, the speedups are not as impressive as usually reported. We deployed what we term the "competing programmer approach" to conduct an as fair as possible performance evaluation. Thus, by adopting competitive spirit to develop the fastest code for each competing platform while investing comparable amount of time on each implementation, we showed in this chapter that state-of-the-art FPGAs (Virtex 6) can be up to 9.65 times faster for computing the phylogenetic parsimony kernel than state-of-the-art CPUs (Intel i7).

Chapter 5

Likelihood Reconfigurable Architectures

This chapter presents four dedicated architectures (four generations) for implementing the phylogenetic likelihood function on reconfigurable logic.

5.1 Introduction

Driven by novel wet-lab sequencing technologies, there has been an unprecedented molecular data explosion over the last few years. Phylogenetic trees are currently increasingly reconstructed from multiple genes or even whole genomes. The recently introduced term "phylogenomics" reflects this development. Hence, there is an urgent need to develop new techniques and computational solutions to calculate the computationally intensive scoring functions for phylogenetic trees.

The phylogenetic likelihood function (PLF) represents one of the most accurate statistical models for phylogenetic inference. While there are many software packages that implement the PLF, either for standard maximum likelihood optimization (RAxML [182], GARLI [4], PHYML [80]) or to conduct Bayesian phylogenetic inference (MrBayes [165], PhyloBayes [112]), the underlying computational problems are identical: all PLF-based phylogenetic inference programs spend the largest part of the overall runtime, typically between 85% and 95%, on the computation of the PLF [147, 182, 191]. Therefore, it is important to assess and devise architectural solutions for this widely used and challenging Bioinformatics function.

In this chapter, we assess via a series of four dedicated computer architectures how the compute- and memory-intensive PLF can be mapped onto reconfigurable hardware and what the performance gains of such specialized PLF architectures are. We exclusively focus on the exploitation of fine-grain parallelism in the PLF, despite the fact that both ML and Bayesian phylogenetic inference also exhibit a source of coarse-grain parallelism. In standard ML analyses, embarrassing parallelism can be exploited via independent tree searches on bootstrap

replicates [66] or through independent parallel searches for the best-scoring ML tree on distinct starting trees [182]. The main reason to focus on fine-grain parallelism is the current trend in Systematics toward analyses of large phylogenomic alignments (see [52, 81, 128] for examples). As already mentioned, such phylogenomic alignments consist of a large number of concatenated genes—a study comprising almost 1,500 genes required 2.25 million CPU hours and 15 GBs of main memory on an IBM BlueGene/L supercomputer [88, 146]—and are extremely memory- and compute-intensive. Therefore, a large amount of computational resources needs to be allocated for concurrently computing the likelihood on a single, large tree topology. Generally, fine-grain loop-level parallelism fits well to the current trend in Systematics, and super-linear speedups (because of increased cache efficiency) can be achieved on large phylogenomic datasets [147, 190, 191]. Finally, parallelization at a fine-grain level naturally maps better to a reconfigurable architecture, while coarse-grain parallelism can be exploited at a higher level, for instance, on a cluster of CPU-FPGA or CPU-GPU nodes.

While there exists a large diversity of methods and software tools for phylogenetic inference, only few have been mapped to hardware. Mak and Lam [122, 123] map a PLF implementation with reduced floating-point precision to reconfigurable logic. The Jukes-Cantor model (JC69 [98]), which is implemented in this work, represents the simplest statistical model of nucleotide base substitution and is rarely used in present-day biological analyses [160]. The performance tests reported in [122] and [123] have been conducted on trees with only 4 leaves (4 input sequences), and scalability beyond 4-taxon trees is not addressed.

Davis *et al.* [49] present an implementation of the simple UPGMA [176] (Unweighted Pair Group Method with Arithmetic Mean) tree reconstruction method. Due to the many simplifying assumptions made in the UPGMA algorithm, it is practically not used for real-world analyses any more.

Bakos *et al.* [29, 30] focus on the reconstruction of phylogenetic trees using gene-order input data, that is, the order of corresponding genes in the genomes of different organisms is used as input data for reconstructing trees. Bakos *et al.* mapped GRAPPA [132], an open-source implementation for gene-order-based phylogenetic inference, onto FPGAs. The main difference to PLF-based phylogenetic inference is that the kernel function used in gene-order analyses is discrete. This means that the amount of floating-point operations required to reconstruct a phylogeny is small, and that a FPGA implementation can mostly rely on integer arithmetics.

Zierke and Bakos [216] also presented a FPGA accelerator for the PLF for Bayesian MCMC-based (Markov chain Monte Carlo) inference methods. The authors mapped the MrBayes [165] PLF implementation for DNA data to reconfigurable hardware. They included numerical scaling techniques (see [183] for details on numerical scaling in the PLF) to prevent numerical underflow as well as a component for calculating log likelihood scores. The speedup estimates (based on the largest available Virtex 6 SX FPGA at the time of publication) that are reported in the paper vary between 2.5 and 8.7 compared to a single state-of-the-art Intel Xeon 5500-series core. Note that Bayesian inference programs do not require numerical optimization routines (e.g., Newton-

Raphson) for branch length optimization, since the MCMC procedure is used to integrate over branch lengths. Hence, the PLF implementation, as used in Bayesian inference programs, is less complex than for ML programs. ML programs typically deploy Newton-Raphson optimization procedures for branch length optimization, which makes hardware design more challenging since dedicated components for computing the first and second derivatives of the likelihood function are also required.

The next four sections (Sections 5.2, 5.3, 5.4, and 5.5) present the four reconfigurable architectures for computing the PLF. For a detailed description of the PLF see Section 2.4.2.3.

5.2 1^{st} generation architecture

Our initial PLF architecture only covers a subset of the operations and functions required to conduct a full real-world tree search as implemented for instance in RAxML [182]. The subset of computations we consider here corresponds to conducting a full tree traversal to score a given, fixed tree topology using Felsenstein's pruning algorithm (see [65] and Section 2.4.2.3).

The 1^{st} architecture takes as input the aligned DNA sequences, computes the entries of the inner probability vectors, and stores them in the internal (embedded) memory of the FPGA. The values of the tip and inner probability vectors are then successively reused to compute all inner probability vectors bottom-up toward the virtual root of the tree. As already mentioned, this 1^{st}—exploratory—architecture only implements a subset of PLF operations. We only conduct full tree traversals from the leaves (tips) toward the virtual root of the tree to compute the likelihood score on a fixed topology with fixed branch lengths and fixed model parameters. In addition, for the time being, we assume that all branch lengths are equal, that is, the transition probability matrix $P(t) = e^{Qt}$, where t is the branch length, is constant across the tree. Also, we have not implemented a scaling procedure for very small probability vector entries since scaling only affects very large trees with hundreds to thousands of sequences. Finally, we have not implemented a model of rate heterogeneity. However, the Γ model [210] of rate heterogeneity can be accommodated by conducting as many tree traversals as there are discrete Γ rates (typically 4 or 8). Note that an appropriate rate-dependent formatting of the P matrices is also required for each tree traversal. Nonetheless, the architecture is based on the computations required for the most complex General Time-Reversible (GTR) [111, 161] model of nucleotide substitution. Finally, the likelihood score can only be computed on fully balanced binary trees of sizes 4, 8, 16, 32, etc.

5.2.1 Design

The block diagram of the proposed architecture is depicted in Figure 5.1. The unit designated as `Likelihood Vector Creator` reads the sequence data and generates a probability vector for every nucleotide by converting the 4-bit word that encodes a DNA character to four double-precision floating-point numbers. The conversion is performed via a

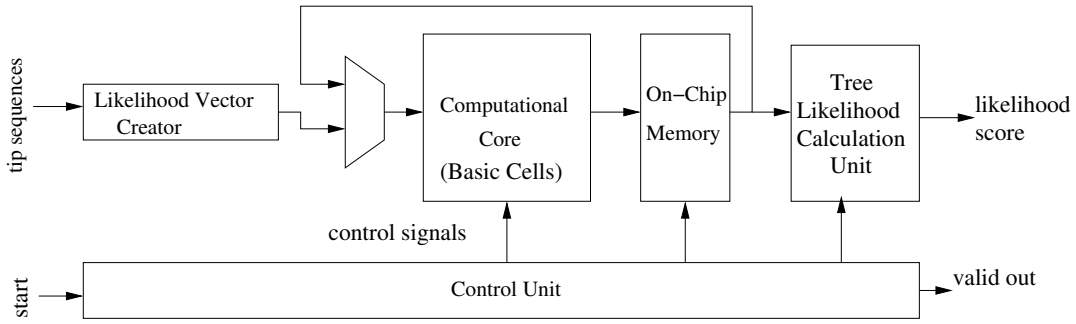


Figure 5.1: Block diagram of the 1^{st} gen. architecture for the phylogenetic likelihood function.

look-up table which complies with the standard definition for ambiguous DNA character encoding (see `http://www.hgu.mrc.ac.uk/Softdata/Misc/ambcode.html`). The output of the `Likelihood Vector Creator` is then forwarded to the `Computational Core`.

The `Computational Core` consists of seven `Basic Cell (BC)` units (see Figure 5.2) that can operate in two distinct modes (see below) and are arranged in a binary tree as shown in Figure 5.3. This tree also reflects the datapath of the system. Each `BC` unit consists of nine multipliers and six adders that are organized as outlined in Figure 5.2 and conducts the main bulk of the likelihood computations. The multipliers and adders in a `BC` are pipelined with a throughput of one result per cycle and a latency of fifteen and fourteen cycles, respectively. Due to the limited amount of DSP48E slices that are available to implement floating-point operations on the FPGA, several multiplexer units are deployed to optimally exploit the computational resources available. Each `BC` has an input steering signal of 1 bit, which is used to select among two basic operations. When a `Basic Cell` works in mode `0`, it calculates the inner conditional probability vector from two child nodes that can either be tip vectors or inner vectors. Mode `1` is invoked when the basic cell needs to calculate the conditional probability vector at the virtual root of the tree. This requires distinct arithmetic operations because there is only one transition probability matrix to be multiplied with the respective probability vector. Remember that the virtual root is placed on a branch right next to one of the two adjacent nodes (see Section 2.4.2.4).

All blocks of embedded memory are organized in eight parts of size 9,216x256 bits. Each of the eight parts is used to store inner probability vectors which may need to be read again—depending on the input tree size—by the `Basic Cells` after several hundreds of cycles. As the computations proceed bottom-up toward the virtual root, those eight parts of embedded RAM are read or written several times depending on the tree depth. Data stored in these memory locations can be overwritten several times, but we maintain a distance of 174 positions (latency of the datapath) between the read index and the write index for each of the eight memory parts.

When the probability vector at the virtual root has been computed by the `Basic Cells` for a set of alignment columns, the `Tree Likelihood Calculation` unit then computes and

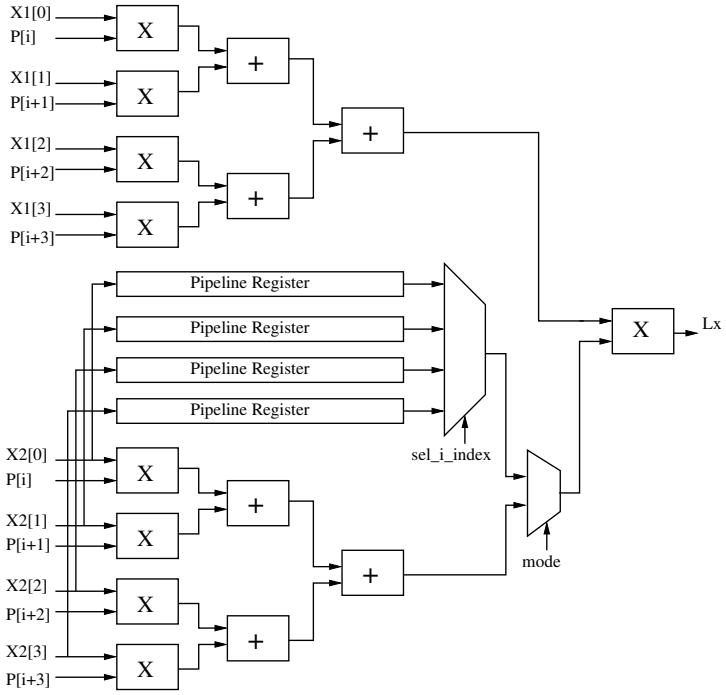


Figure 5.2: Arrangement of multipliers and adders within a Basic Cell unit. Vectors $X_1[]$ and $X_2[]$ represent the probability vector entries of the right and left child, while $P[]$ represents a single row of the transition probability matrix $P(t)$. The mode multiplexer is used to select among functions for computations at the virtual root and at inner nodes below the virtual root. The output L_x corresponds to one of the 4 probability values $P(A)$, $P(C)$, $P(G)$, and $P(T)$ that are written to the probability vector above the input vectors.

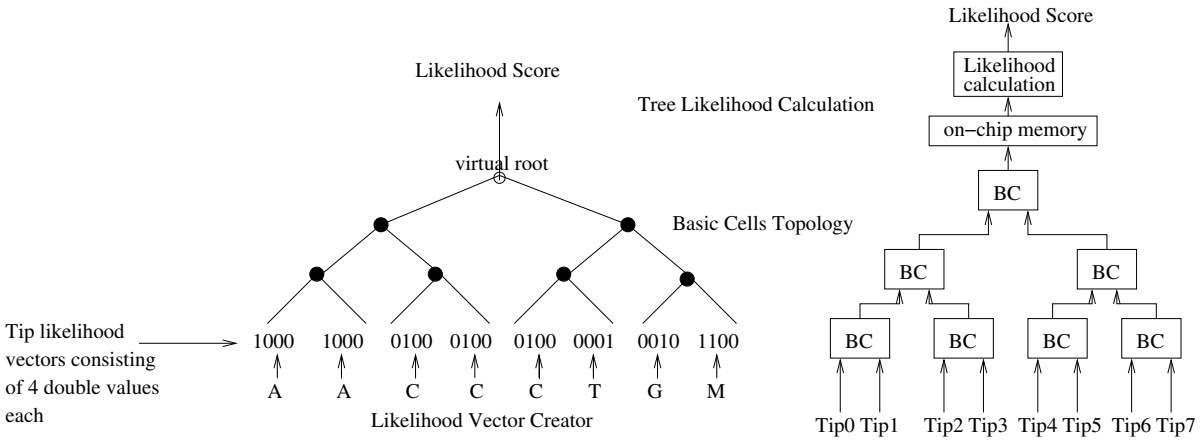


Figure 5.3: Example for the computation of the likelihood score on a single alignment column for an 8-taxon tree using the Likelihood Vector Creator, Basic Cell, and Tree Likelihood Calculation units.

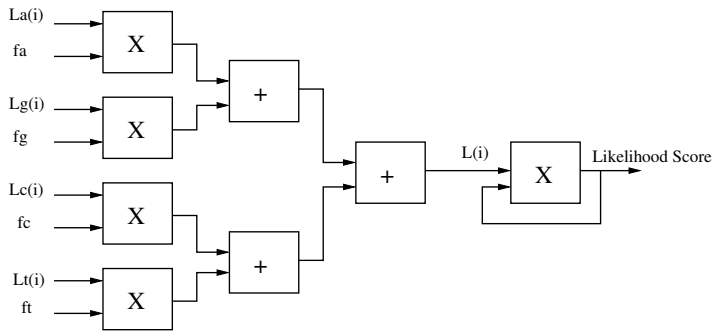


Figure 5.4: Tree Likelihood Calculation unit: computation of likelihood score $L(i)$ for a single alignment column at position i and accumulation of likelihood scores over several alignment columns $i = 1...n$. This unit takes as input the inner conditional probability vector entries $La(i)$, $Lc(i)$, $Lg(i)$, $Lt(i)$ (denoted as Lx here since they are computed with the Basic Cell in Mode 1) at position i of the virtual root and the base frequencies (prior probabilities) fa, fc, fg, ft (also denoted as $\pi_A, ..., \pi_T$) and multiplies the result $L(i)$ with the product of the previous results $L(0) \cdot L(1) \cdot ... \cdot L(i-1)$ such that $L = L(0) \cdot L(1) \cdot ... \cdot L(i)$.

combines the per-site (per-column) likelihood scores and returns the overall likelihood score of the tree. The basic components of this unit are illustrated in Figure 5.4. Finally, the `Control Unit` consists of a hierarchy of five FSMs (Finite State Machines) which concurrently control the datapath. We use a master FSM to synchronize the four remaining worker FSMs. The worker FSMs are used to generate the required control signals.

5.2.2 Operation

The architecture operates in three phases. During the *first phase*, the system reads the nucleotide sequences at the tips and initially translates them into probability vectors. Then, it calculates the probability vectors of their common ancestors bottom-up, up to three levels above the tips (see tree of BCs in Figure 5.3), and stores the results in embedded RAM. This *first phase* (computation of inner probability vectors for 8-taxon trees) is successively executed for the entire input tree in groups of 8 taxa (e.g., 4 times for a 32-taxon tree). The first phase is completed when the entire input tree and data have been read, and the probability vectors of the ancestors of the respective 8-taxon trees have been computed (e.g., 4 conditional probability vectors in the 32-taxon case).

The *second phase* is very similar to the initial phase. However, instead of using the DNA sequence information at the tips of the tree as input, it uses the probability vectors of the ancestors (previously computed internal nodes of the tree) that have been stored in embedded RAM during the first phase. The results of the computations of the second phase (if the virtual root of the tree has not already been reached) are then stored again in embedded RAM. *Phase 2* is repeated for a maximum of 8 internal nodes at a time (if the number of taxa is a power of 8)

until all probability vectors at the input level have been used and the inner probability vectors three levels above the input level have been computed. At the end of *phase 2*, the conditional probability vector at the virtual root has already been calculated and is available for computing the overall likelihood.

In the *third* and final *phase*, the system loads the probability vector at the virtual root from embedded RAM and calculates the per-site likelihood score $L(i)$ for every column of the input alignment. A multiplier is used to compute the product $L = L(0) \cdot L(1) \cdot ... \cdot L(n-1)$ over all n per-site likelihood scores. Note that the `Tree Likelihood Calculation` unit actually computes the likelihood score and not the more commonly used log likelihood score. If the input data has 8 taxa or less, *phase 2* is omitted since the internal probability vector three levels above the input sequences already *is* the virtual root of the tree. For a 64-taxon input tree, *phase 1* will be executed 8 times and will produce 8 internal probability vectors. Then, *phase 2* will be executed only once for the resulting 8 probability vectors and yield the probability vector at the virtual root. Analogously, for a 512-taxon tree, *phase 1* will be executed 64 times, *phase 2* will be executed 8 times for the first set of internal probability vectors three levels above the tips, and 1 more time for the calculation of the probability vector at the virtual root.

5.2.3 Experimental setup and results

To conduct performance analyses, we generated 8 simulated DNA datasets containing 4, 8, 16, 32, 64, 128, 256, and 512 taxa (sequences/organisms), respectively. Every simulated alignment contains 1,000 distinct alignment columns, that is, the alignment can not be further compressed by merging identical columns into site patterns. In addition, we generated respective fully balanced, binary input trees in standard NEWICK format. The datasets and the software implementation are available at `http://www.exelixis-lab.org/software/FPGA-DATASETS.tar.bz2` and `http://www.exelixis-lab.org/software/RAxML-FPGA.tar.bz2`, respectively.

To ensure a fair comparison, as reference software implementation we used a significantly strapped down version of RAxML that executes exactly the same mathematical operations as the computational kernel on the FPGA. The sequential as well as the OpenMP-based versions of the code were compiled using the Intel `icc` compiler (version 10.1, optimization option `-O3`), which generates faster code than `gcc` for RAxML. The loop-level OpenMP parallelization of RAxML was re-implemented as described in [190]. In order to obtain accurate software timing results for the very fast tree traversal operation that takes less than 0.015 seconds on all datasets, we measured the time required by a loop that executes 20,000 full tree traversals. As software test platform we used a high-end 8-way dual-core SUN x4600 system equipped with 64 GBs of main memory and 8 dual-core AMD Opteron processors running at 2.6 GHz. On long enough input alignments and hence long enough for-loops, we measured super-linear speedups for both Pthreads- and OpenMP-based parallel implementations of the standard RAxML running on all 16 cores [190]. The OpenMP-based strapped down version of RAxML was executed using 2, 4, 8, and 16 cores. As HW platform we used the Xilinx Virtex 5 SX240T. This FPGA provides a

large number of DSP48E slices (1,056). The DSP48E slices are used to implement the double-precision floating-point multipliers and adders that form part of the `Basic Cells`. The specific FPGA offered the largest number of embedded memory blocks available in commercial FPGAs at the time the experiments were conducted and was hence well-suited for our purpose.

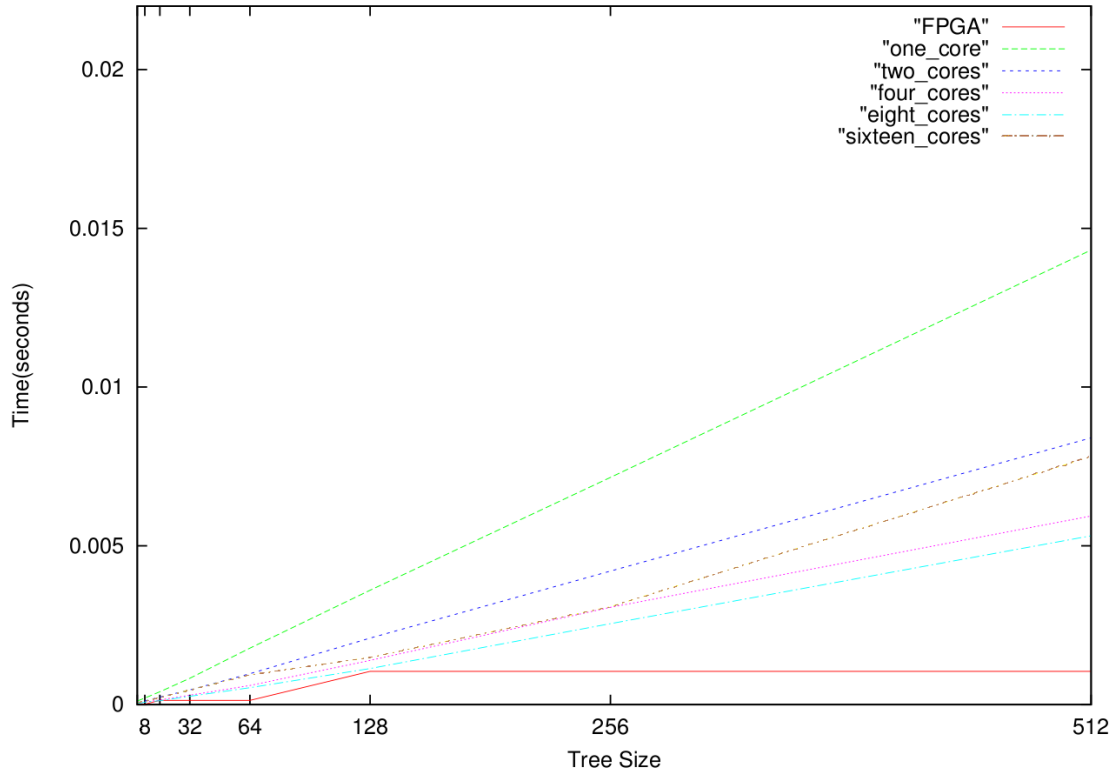


Figure 5.5: FPGA and multi-core execution times for full tree traversals using Felsenstein's pruning algorithm for trees with 4, 8, 16, 32, 64, 128, 256, and 512 taxa.

To validate the correctness of the results (likelihood scores) returned by the FPGA implementation, we deployed an accurate cycle-by-cycle, post-place-and-route simulation for datasets with 256 alignments columns and 8, 64, and 512 taxa. The FPGA yields *exactly* the same results as the software implementation. In order to measure execution times for the FPGA implementation, we used the static timing report of the Xilinx tools (ADVANCED 1.53 speed file). The reported clock speed was 284 MHz. According to this clock speed and the number of cycles that the system requires to perform likelihood calculations, we calculated projected execution times on the FPGA (see Figure 5.5).

The results outlined in Figure 5.5 depict the execution times of one full tree traversal on the FPGA and of the software implementation on 1, 2, 4, 8, and 16 cores on the Sun x4600

multi-core system. The slowdown that can be observed for the OpenMP version with 16 cores is clearly due to the relatively short alignment length and hence an unfavorable communication-to-computation ratio. FPGA performance improves slightly with increasing dataset size. As depicted in Figure 5.5, the execution times on the FPGA for 4- and 8-taxon trees, for 16-, 32-, and 64-taxon trees, as well as for 128-, 256-, and 512-taxon trees are identical. This behavior is related to the degree of system utilization. If the number of taxa in the input tree is a power of 8, the `Basic Cells` will be fully utilized at all times during the entire computation until the virtual root is reached. If the number of taxa is for instance 4, the same number of cycles as for an 8-taxon tree will be required to reach the virtual root, but 50% of the `Basic Cells` will not be used. Thus, the execution times will remain constant for $8^n < t \leq 8^{n+1}$, where t is the number of taxa in the input tree. Thus, the computational resource utilization in our design is optimal when the input comprises 8, 64, or 512 taxa.

# Cores	4 taxa	8 taxa	16 taxa	32 taxa	64 taxa	128 taxa	256 taxa	512 taxa
1	6.81	12.76	3.07	6.30	13.55	3.44	6.83	13.68
2	3.79	7.35	1.79	3.53	7.39	2.00	4.02	8.03
4	2.32	**4.74**	1.15	2.26	4.60	1.33	2.93	5.67
8	**2.09**	4.85	**0.96**	**2.01**	**4.05**	**1.08**	**2.44**	**5.08**
16	3.65	6.82	1.68	3.55	7.14	1.42	2.95	7.46

Table 5.1: FPGA speedups on 4- up to 512-taxon trees compared to software execution times on 1 up to 16 cores. The worst speedup achieved by the FPGA on the respective datasets is indicated by bold letters.

Table 5.1 shows the speedup values that the FPGA achieved compared to the multi-core system on 1, 2, 4, 8, and 16 cores/threads for all tree sizes. The respective worst speedups obtained by the FPGA is shown in bold letters. In all but one cases (16 taxa, 8 cores), the FPGA is faster than the high-end multi-core system executing a highly optimized PLF implementation that has been compiled with an efficient commercial compiler for this type of application. While the worst speedup achieved is 0.96 (this slowdown is negligible), this only occurs in the most unfavorable case for the FPGA implementation: a large number of `Basic Cells` is under-utilized (because of the size of the tree).

5.3 2^{nd} generation architecture

In this section, we present the 2^{nd} PLF architecture, which significantly extends the initial proof-of-concept design of Section 5.2. The initial architecture was only able to compute the PLF on fully balanced trees, but it already yielded a substantial performance boost. The major extension in the more versatile architecture presented here is: i) it can evaluate the PLF for any given tree topology at the same speed as the previous architecture, ii) it exploits the intrinsic

parallelism of the PLF in a more flexible and scalable way, and iii) it is able to conduct so-called partial tree traversals which represent a fundamental mechanism in current tree-search algorithms. In this architecture, input/output issues have been taken into account to allow for mapping it to a modern platform.

Felsenstein's pruning algorithm [65] is the standard method to compute the PLF and hence the likelihood score for a given tree topology. In the following, we provide an abstract description of this algorithm. The first step consists in identifying/locating a pair of child nodes i and j in the given tree for which the probability vector at the common ancestor k ($1 \leq i, j, k \leq 2n-2$) has not already been computed. The second step is to calculate the probability vector entries of that common ancestor (ancestral probability vector at k) and prune the child nodes from the tree. These steps are executed recursively until the probability vector at the virtual root vr has been calculated, and thus the pruning process has transformed the initial tree to only one node that is located at the virtual root. Remember that phylogenetic trees under ML are unrooted for mathematical and computational reasons (Section 2.4.2.3), but also keep in mind that a virtual root can be placed into any branch of the tree to evaluate its likelihood score.

We propose a master-worker architecture that consists of two main units: the Target Pair Unit (*TPU*, master) and the Computational Basic Core (*CBC*, worker). The *TPU* (master) performs the first (pruning) of the previously described algorithmic steps, while the *CBC* (worker) unit performs the second (calculating). The *TPU* executes the tree traversal steps of the pruning algorithm on the contents of a local memory that is used to hold information about the tree structure. The *TPU* tracks down which tips and/or inner nodes should be combined to compute the entries of an ancestral probability vector. The information needed by the worker to locate the tip sequences or the probability vectors in the external or internal memories and start calculating the ancestral probability vector is provided/communicated via shared registers. Both the *TPU* as well as the *CBC* have access to these registers. Once the *TPU* has written the required addresses and selection bits, it sends a start signal to the *CBC*, which indicates that there are valid data available in the registers. This means that the *CBC* can start the calculation process. At the same time, the *TPU* switches to stand-by mode. The *TPU* now waits for the *CBC* to calculate the ancestral probability vector and write back (to the shared registers) the address and selection bits of the memory position where the newly computed ancestral probability vector has been stored. The *CBC* then announces that the calculation process has been completed by sending a respective signal to the *TPU*. The *TPU* will then update its local memory accordingly, using the information from the shared registers, and will determine the next pair of tips and/or nodes for which an ancestral vector needs to be computed. We denote this design as master-worker architecture because only the *TPU* can initiate computations on the *CBC*. The general architectural scheme of the design is illustrated in Figure 5.6.

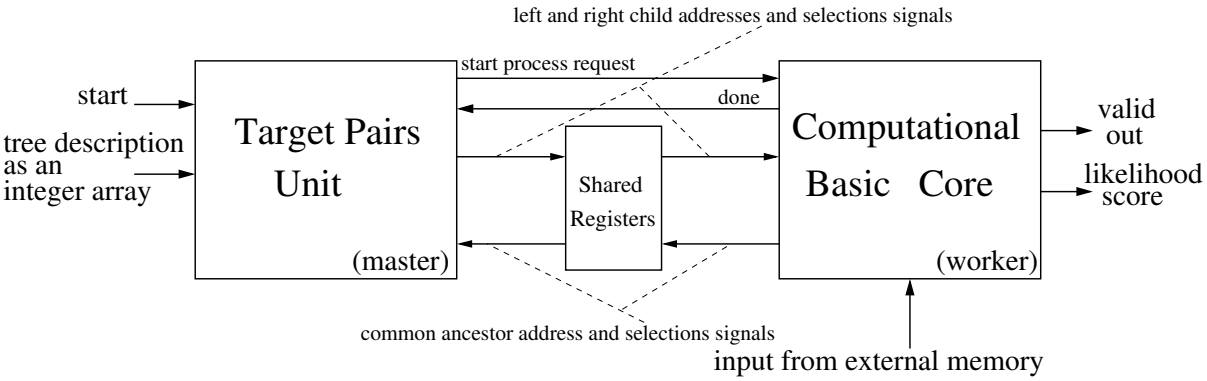


Figure 5.6: Architecture overview.

5.3.1 Pruning unit

As already mentioned, the *TPU* executes Felsenstein's pruning algorithm on the contents of its local memory. The top-level architecture of this unit is outlined in Figure 5.7. The unit consists of three structural components, a *Control Unit* implemented as FSM, and a local memory.

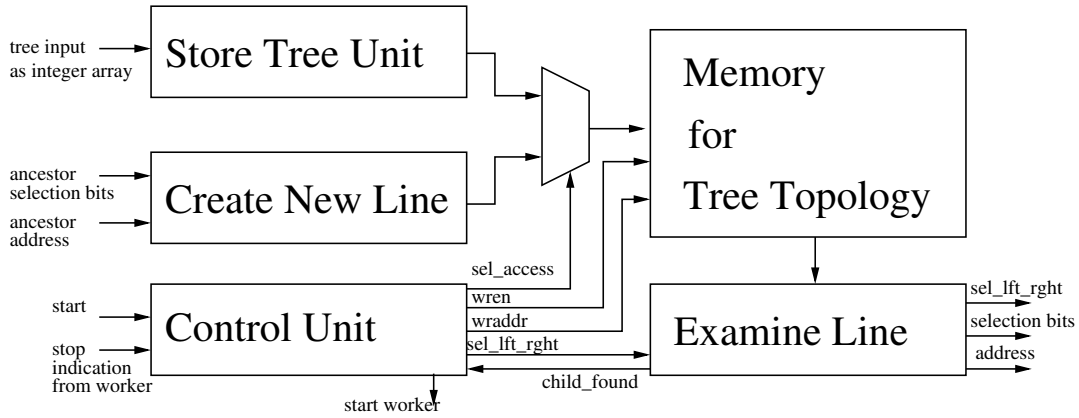


Figure 5.7: Architecture of the Target Pair Unit (TPU).

The component *Examine Line* in conjunction with the *Control Unit* execute the algorithm to determine the pairs of tips and/or nodes to be combined. The *Store Tree* unit is used to initialize the contents of the local memory. It reads in an integer array that describes the tree topology to be evaluated. This integer array contains the node depths (distances from the virtual root) in depth-first order from the virtual root. For example, the tree illustrated in Figure 5.9 (top right) is given by the integer array: 1 (depth of A), 2 (depth of D), 3 (depth of C), 3 (depth of B). The *Create New Line* unit is used to back-transfer the information that the *CBC* has written to the shared registers. This information describes the ancestral vector that

has just been calculated.

The fields in the memory lines of the local memory that holds the tree structure are shown in Figure 5.8. The valid field indicates whether the memory line contains useful information for the remainder of the pruning process. The *depth* field contains the depth of the tip or

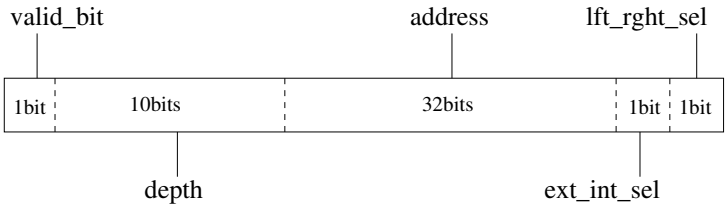


Figure 5.8: Layout of a tree topology memory line.

node that the line describes. As already mentioned, the depth represents the depth-first node distance to the virtual root, for instance, the two child nodes of the virtual root have depth one. The remaining fields (*address*, *ext_int_sel*, and *lft_rght_sel*) hold the necessary information for addressing the nucleotide sequences at the tips or the ancestral probability vectors. The address field is an index to the memory line that contains the first nucleotide of a DNA sequence (when the line denotes a tip) or the first probability vector entry (when the line represents an inner node). Both the nucleotide sequences and the ancestral probability vectors are stored contiguously in the external and the internal memories, respectively. The address field contains a memory address, but it has not yet been specified whether this address refers to internal or external memory. This information is provided by the *ext_int_sel* (external/internal selection) field. If this bit is set, the address refers to the internal memory. Internal memory is organized into two big parts to reduce complexity of controlling simultaneous read (from internal memory) and write (data coming from external memory) operations. The *lft_rght_sel* (left/right selection) field is used to select between these two parts. If this bit is set, the address refers to the right part of the internal memory. Each valid line represents a tip or a node vector of the tree.

Two snapshots of the local memory that illustrate how the *TPU* works are depicted in Figure 5.9. For the four-taxon tree shown in the figure (top right), the master unit initializes the memory as illustrated by the table in the top left corner. Every tip of the tree has been labeled by a capital letter. The virtual root has been placed into the branch that connects A to the rest of the tree. There is also one number for every node of the tree (tips as well as internal nodes) that indicates the distance to the virtual root. The component *Examine Line* reads the memory from the top and stores the contents of the fields *address*, *ext_int_sel*, and *lft_rght_sel* in the shared registers. Then, the *Control Unit* triggers the *CBC* to start the calculation. When the *CBC* has calculated the respective probability vectors, it updates the shared registers with the address and selection bits of the ancestral vector, which has been labeled by E in the second tree of Figure 5.9 (lower part). Finally, the unit *Create New Line* updates the memory, and the

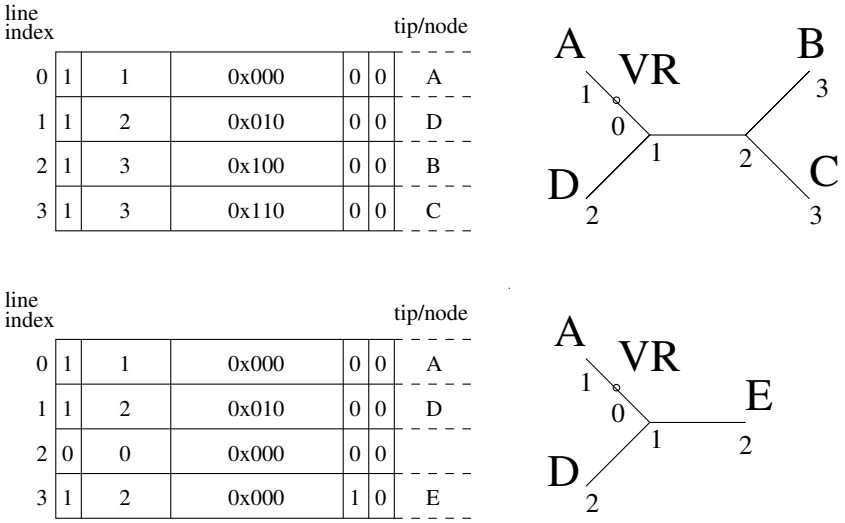


Figure 5.9: Two consecutive snapshots of the tree topology memory and the respective trees.

contents of it become available to the next step of the pruning algorithm. The memory update sequence is thus equivalent to the pruning steps in the tree.

5.3.2 Pipelined datapath

Initially, the *CBC* (worker unit) remains idle while the *TPU* determines the nodes to be combined. When the *CBC* is triggered by the *TPU* to start the calculations, it fetches the data at the position provided by the address and selection bits in the shared registers and initiates the calculation of the probability vectors. An overview of the *CBC* architecture is provided in Figure 5.10. It consists of the *Control FSM*, the *Basic Cell Array*, the *Fetch Units* (FIFOs), internal memories, and the *Likelihood Score Unit*.

The *Fetch Units* have been specifically designed to hide the latency of linear external memory accesses to probability vectors (evidently the latency can not be hidden for the initial accesses to vector or tip addresses). Since the probability vectors are long (typically $m > 1,000$), the latency for accessing the first datum of an array is negligible, while we can achieve infinitely large burst ability. The *Basic Cell Array* consists of ten *Basic Cells* (see Figure 5.2) which perform the double-precision floating-point additions and multiplications provided in Equation 2.9. All *Basic Cells* work in parallel on a different column of the sequence alignment that is stored in external memory. This design can easily be extended to a maximum of m *Basic Cells*, and every *Basic Cell* can work concurrently to compute one of the entries of the respective vector L of length m. According to the node pair provided by the *TPU*, the appropriate data are prefetched and stored in the *Fetch Units* or accessed directly in internal memories. The resulting ancestral probability vector is written to the internal memories. Once the probability vector of the virtual root has been computed, the *Basic Cells* are used again to calculate the per-column likelihood

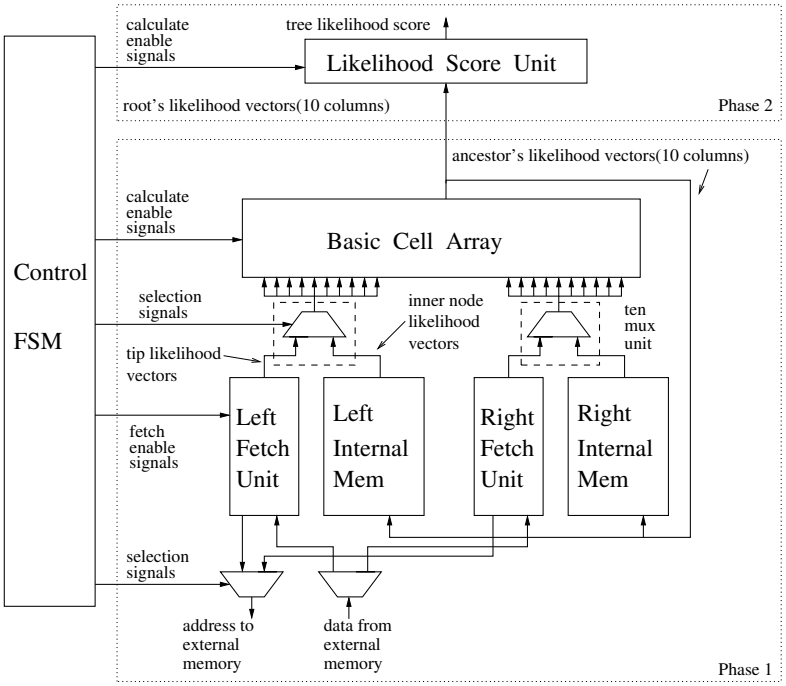


Figure 5.10: Overview of the Computational Basic Core (CBC).

scores (see Equation 2.10), and the product over the per-column scores $l(i)$ is then computed by the *Likelihood Score Unit*.

The *Basic Cell Array* consists of 10 *Basic Cells* that work in parallel. Both the 1^{st} and the 2^{nd} generation PLF designs employ the same *Basic Cell* architecture. Each *Basic Cell* is arranged as a tree of double-precision floating-point adders and multipliers which also operate in parallel as shown in Figure 5.2. The *Basic Cells* are fully pipelined with a total pipeline depth of 58 cycles. Each *Basic Cell* evaluates one probability value per cycle, generating one probability vector entry output every 4 cycles (for all 4 nucleotides). Because the operations required for the calculation of the probability vector at the virtual root are slightly different, the *Basic Cell* also contains a vector of pipeline registers and a 4-to-1 multiplexer to perform the appropriate operations when the respective mode signal is set (see Section 5.2.1).

5.3.3 Evaluation and performance

Extensive post-place-and-route simulations were conducted to verify the functionality of the proposed architecture. The input datasets again contained 1,000 distinct alignment columns and trees with 4, 8, 16, 32, 64, 128, 256, and 512 taxa (available at `http://www.exelixis-lab.org/software/FPGA-DATASETS.tar.bz2`). The results (likelihood scores of the trees) computed by the architecture were exactly identical to those obtained by the software implementation (light-weight version of RAxML, available at `http://www.exelixis-lab.org/software/`

`RAxML-FPGA.tar.bz2`). As before, we executed the program with 1, 2, 4, 8, and 16 threads on the high-end SUN x4600 system, and measured the time required by a loop that executes 20,000 full tree traversals. The proposed architecture was mapped to a Xilinx V5 SX240T FPGA. The design with 10 parallel *Basic Cells* uses 87% of the slice LUTs, 94% of the Block RAMs (BRAM slices), and 93% of the DSP48E slices on this device.

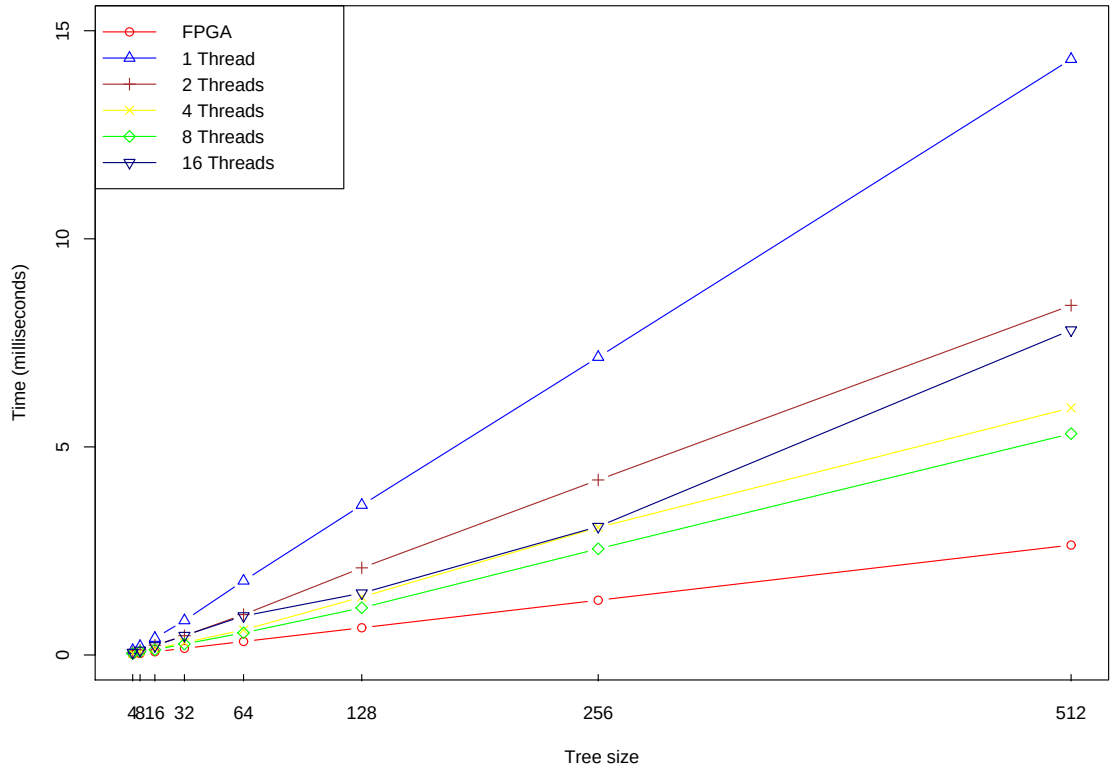


Figure 5.11: FPGA versus multi-threaded execution times on the Sun x4600.

The clock speed that was measured for the design amounts to 101 MHz (static timing report of the Xilinx Tools, ADVANCED 1.53 speed file). Figure 5.11 provides the projected FPGA execution times and the actual software execution times on the Sun x4600 for 1, 2, 4, 8, and 16 threads as a plot over input tree size for the respective datasets, and Table 5.2 shows the speedups. As can be observed in the plot, the execution of the software (SW) implementation with 16 threads is slower than with 8 threads. The reason for this slowdown and the lack of scalability is an unfavorable communication-to-computation ratio. For all test datasets, FPGA performance is better than that of the highly optimized SW implementation running in

Taxa	1 Thr.	2 Thr.	4 Thr.	8 Thr.	16 Thr.
4	7.04	3.92	2.40	2.16	3.77
8	5.76	3.32	2.14	2.18	3.08
16	5.22	3.04	1.95	1.62	2.84
32	5.19	2.90	1.86	1.66	2.92
64	5.49	3.00	1.86	1.64	2.89
128	5.50	3.20	2.18	1.73	2.27
256	5.44	3.20	2.33	1.94	2.34
512	5.41	3.18	2.28	2.01	2.96

Table 5.2: Speedups of the hardware design compared to the multi-core software implementation.

parallel on a high-end multi-core machine. Despite the fact that the execution times are within the millisecond range, in real application scenarios search algorithms will invoke likelihood computations millions of times to conduct ML estimates of model parameters and to search for the best ML tree topology.

5.4 3^{rd} generation architecture

Here, we present a dedicated computer architecture for a PLF co-processor that is sufficiently generic to compute likelihood scores on all common input data types: morphological (binary), DNA, RNA secondary structure, and protein data. The architecture we present here is the 3^{rd} generation architecture for the PLF on reconfigurable logic. The previous two generations (Sections 5.2 and 5.3) were only able to execute a small fraction of the required PLF routines required for a phylogenetic analysis with RAxML. This 3^{rd} generation design provides all functions that need to be offloaded to a co-processor by a real-world maximum likelihood program.

A significant improvement over previous designs [122, 123, 216] is that we can accommodate rate heterogeneity. Rate heterogeneity models accommodate the biological fact that different alignments columns/sites in the input alignment evolve at different speeds (see Section 2.4.2.2). The branch length optimization process (briefly described in Section 2.4.2.4) in RAxML is carried out using the Newton-Raphson procedure (see [156] for a summary). The computationally challenging part of branch length optimization consists of calculating the first and second derivatives of the likelihood function. Branch length optimization accounts for approximately 30% of the total execution time of RAxML. To this end, a dedicated architectural component has been designed and integrated into the co-processor for computing the derivatives of the PLF. Moreover, the architecture is also able to calculate transition probability matrices $P(b)$ (see [183] for details) for a given branch length b, and a resource-efficient LAU (Logarithm Approximation Unit [10]) is used to calculate log likelihood scores. Finally, an optimized numerical scaling unit

has been integrated to prevent numerical underflow in the PLF on large trees with many taxa.

5.4.1 Co-processor design

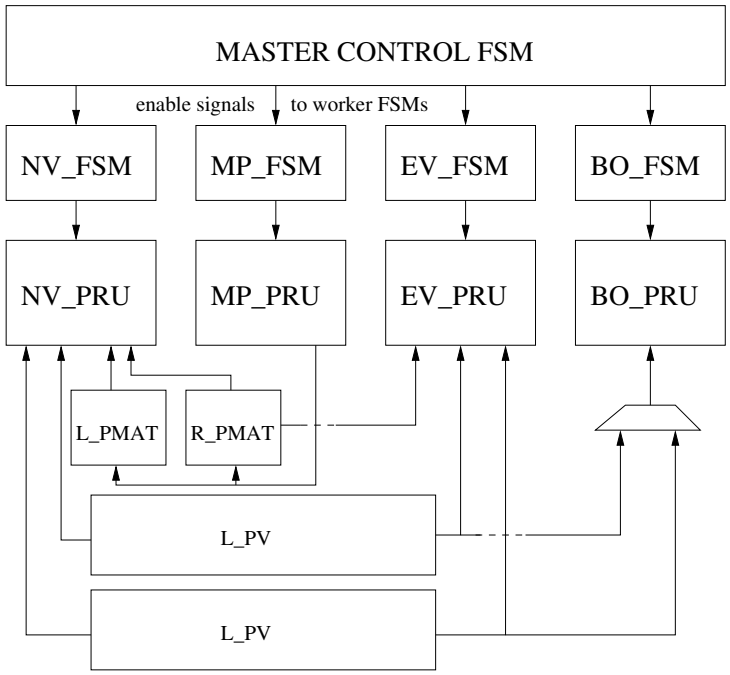


Figure 5.12: Top-level design of the co-processor architecture.

In the following, we describe the architecture of the generic, reconfigurable PLF co-processor. Figure 5.12 provides an abstract view of the design. RAxML [182] was used as reference for the design of the architecture. Thus, each processing unit (PRU) represents a hardware-optimized implementation of a RAxML likelihood function.

The *MASTER CONROL FSM* is located at the top of Figure 5.12. This FSM is vital because it is the only component that can initiate or terminate the operation of the co-processor. The master FSM communicates directly with the worker FSMs, which are placed below the *MASTER CONROL FSM* in Figure 5.12, and coordinates the correct operation of the PRUs. Each PRU is controlled by one worker FSM. The worker FSMs control the components (counters and comparators) that generate read and write addresses for the memory blocks of the co-processor's memory subsystem. In addition, each worker FSM generates appropriate control signals for synchronizing the operation of the respective PRU with the data-fetch and write-back operations.

The co-processor comprises the following PRUs: *NV_PRU* (calculation of the ancestral probability vector), *MP_PRU* (calculation of the transition probability matrix), *EV_PRU* (calculation of the log likelihood score), and *BO_PRU* (calculation of the first and second derivatives

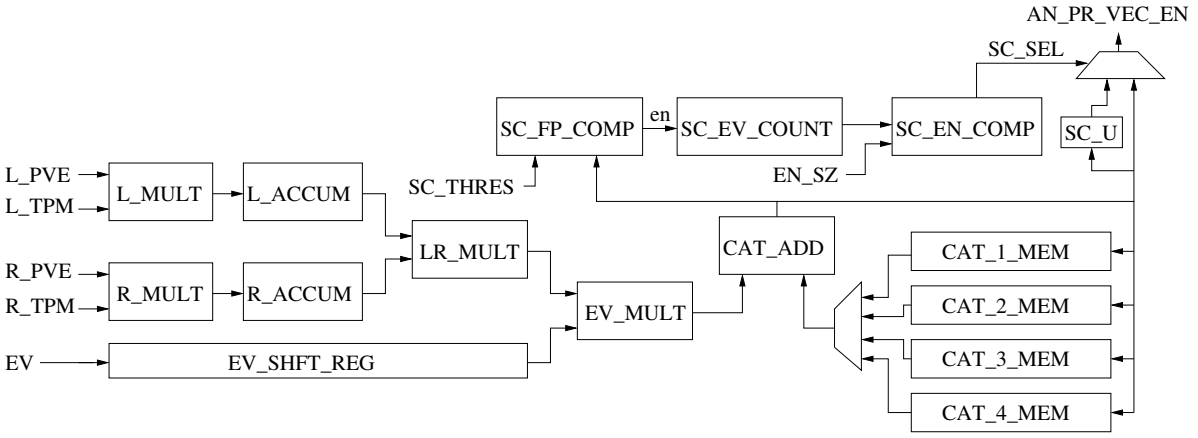


Figure 5.13: Block diagram of the NV_PRU processing unit.

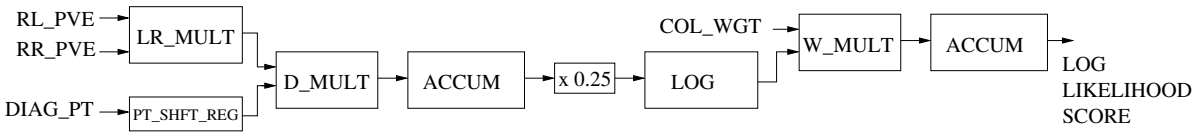


Figure 5.14: Block diagram of the EV_PRU processing unit.

of the PLF). The *NV_PRU* (Figure 5.13) executes the variable-size (variable sizes are required for input data types with a different number of states) matrix-vector multiplication (see Equation 2.9) to compute ancestral probability vectors (including rate heterogeneity and numerical scaling). In Figure 5.13, the left and right child vector entries are denoted as *L_PVE* and *R_PVE*, while the respective left and right P matrices are denoted as *L_TPM* and *R_TPM*. The two multiply-accumulate components (L_MULT→L_ACCUM, R_MULT→R_ACCUM) implement the generic PLF architecture, that is, an architecture that can compute the likelihood on morphological, DNA, RNA secondary structure, and protein data. The *CAT_ADD* adder and the four *CAT_X_MEM* memory components allow for repeatedly using the pipelined datapath. When a Γ model with four discrete rate categories is used, the datapath is traversed four times for each discrete rate. Finally, the numerical scaling subsystem is depicted at the top right of Figure 5.13. The comparator and counter components, denoted as *SC_FP_COMP* and *SC_EV_COUNT*, count the number of values that need to be scaled in the probability vector entry (values smaller than a scaling threshold *SC_THRES*). Then, the *SC_EN_COMP* comparator decides whether the entire probability vector entry needs to be scaled before the write-back operation. The *SC_U* unit scales all entries by multiplying double-precision machine numbers with a constant that is a power of 2 (implemented by an addition in the exponent field). The final 2-to-1 multiplexer then selects the scaled or the unscaled probability vector

entry according to the signal coming from the *SC_EN_COMP* comparator.

When the *NV_PRU* has completed the PLF computations, the *EV_FSM* can trigger the operation of the *EV_PRU* (Figure 5.14) which calculates the overall log likelihood score for the tree at the virtual root. *RL_PVE* and *RR_PVE* are the probability vectors to the left and the right of the branch where the virtual root is located. The accumulator calculates the sum of the likelihoods of the 4 discrete Γ rates. The *x0.25* multiplier that is connected to the accumulator's output bus is required because the Γ model assumes that all 4 discrete rates are equally probable. The *LOG* unit calculates the log likelihood score of the specific column/site. Because alignment columns with equal site patterns can be compressed into a single column for saving on computations, the log likelihood of each column is multiplied by a integer weight (*COL_WGT* in Figure 5.14) that corresponds to the number of identical columns that have been compressed into a single site pattern. Finally, the accumulator sums over the per-site log likelihoods to obtain the overall score for the tree.

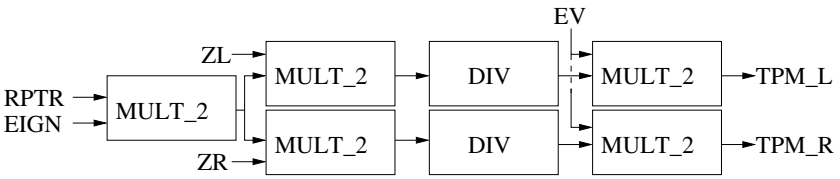


Figure 5.15: Block diagram of the MP_PRU processing unit.

The values of the transition probability matrices P (*L_TPM* and *R_TPM* in Figure 5.13) are calculated by the *MP_PRU* (see Figure 5.15). The dimensions of the P matrices are not constant since they depend on the data type being analyzed; their size can vary between 2×2 (morphological data) and 20×20 (protein data).

Finally, the *BO_PRU* (Figure 5.16) is used to compute the first and second derivatives of the likelihood function. The derivatives are required for the Newton-Raphson procedure that optimizes branch lengths.

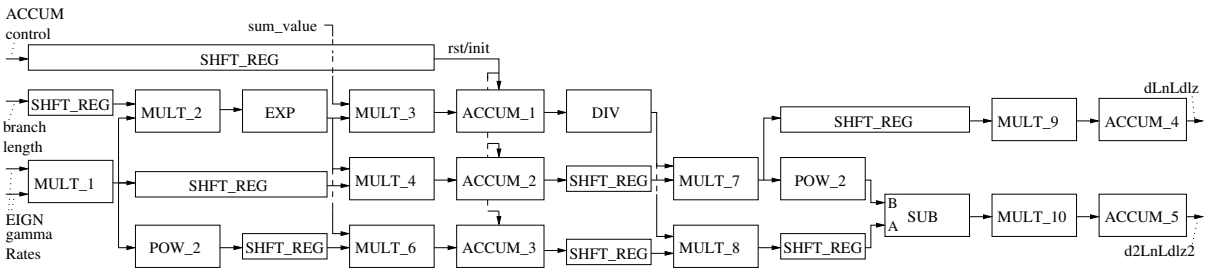


Figure 5.16: Block diagram of the BO_PRU processing unit.

5.4.2 Evaluation and performance

To verify the functionality of the PLF co-processor architecture, we implemented it in VHDL and mapped it onto a Xilinx Virtex 5 SX95T-1 FPGA. We used Modelsim 6.3f by Mentor Graphics as simulation tool and conducted extensive post-place-and-route simulations for each processing unit. For testing, we executed RAxML on real-world datasets and created testbenches for the FPGA implementation by storing the input arguments for the functions implemented in the co-processor. Then, we compared the results calculated by the co-processor with those of the corresponding RAxML functions.

The majority of floating-point operations is executed on the FPGA by components that have been generated using the Xilinx Floating Point Operator (Xilinx Floating Point Operator version 4.0, `http://www.xilinx.com/support/ip_documentation/floating_point_ds335`, accessed July 2009). However, the accumulator and exponential operators have been generated using FloPoCo [51] because the Xilinx Floating Point Operator does not provide these operators. To compute logarithms, we used the Logarithmic Approximation Unit (LAU [10]).

Resources/Performance	NV_PRU	MP_PRU	EV_PRU	BO_PRU	SYSTEM
Number of Slice Registers (58,880)	6,147	4,951	5,885	5,110	22,077
Number of Slice LUTs (58,880)	4,271	6,629	4,841	14,040	31,354
Occupied Slices	1,850	2,163	2,087	3,829	10,780
Number of BlockRAM/FIFO(36k)	0	7	3	3	21
Number of BlockRAM/FIFO(18k)	0	0	1	0	1
Number of DSPs	43	68	39	108	258
Maximum Frequency(MHz)	211	104	223	59	61

Table 5.3: Resource usage and performance report of the processing units on a Virtex 5 SX95T FPGA.

Table 5.3 provides resource usage and performance data for each PRU as well as for a light-weight system that contains only one PRU of each type. As shown by Table 5.3, the entire co-processor design occupies only a fraction of the available reconfigurable resources on the chip. We did not attempt to exploit the full HW capacity of the FPGA since our objective is to present and verify a general computer architecture for the PLF.

The performance of this generic co-processor architecture which can operate on all types of biological input data is affected by two factors: i) the use of reconfigurable—flexible—hardware and ii) the design of a generic, flexible but nonetheless static design on programmable hardware. Our vision was to create a prototype design for an ASIC. Therefore, a static architecture that can handle all types of biological data is required. To this end, the current FPGA implementation of the co-processor is between 2.8 and 4.5 times slower than the highly optimized, SSE3-vectorized PLF implementation in RAxML when executed on an Intel processor running at 2.4 GHz. However, a dedicated accelerator-oriented version of our co-processor that can only handle

DNA data (see Section 5.5) showed a speedup of factor 6 on a Xilinx Virtex 5 SX240T. To allow for reproducing our results, the VHDL source code of the co-processor is available at `http://www.exelixis-lab.org/genericPLF_FPGA.tar.bz2`.

A question that may arise is why runtime reconfiguration is not deployed to improve performance. An initial assessment indicates that, because of the high frequency of reconfigurations required to make available optimized, data-type-specific PRUs on the co-processor, execution times will be largely dominated by reconfiguration times. For instance, RAxML requires 6,743 seconds to complete a tree search on a real-world dataset comprising 714 sequences with a length of 1,231 nucleotides each. The total number of PLF function invocations that would need to be offloaded to the co-processor during the complete tree search amounts to 61,092,096. This corresponds to 9,060 reconfigurations per second. We used the Partial Reconfiguration Cost Calculator (PRCC, `http://users.isc.tuc.gr/~kpapadimitriou/prcc.html`, accessed October 2010) to estimate the reconfiguration time for the entire chip (based on the equation presented in [153]), assuming an average-size FPGA such as the Virtex5 FX200T and a PowerPC as reconfiguration controller. The PRCC tool calculated a reconfiguration time of 576 milliseconds, which means that 9,060 reconfigurations (the number of reconfigurations that would be required per second) will take approximately 5,218 seconds.

Thus, given the current state of technology, a system supporting runtime reconfiguration will not outperform the static generic design we present here. Therefore, our generic design can serve either as a prototype architecture to build an ASIC, or as a starting point to derive accelerator-like FPGA implementations that are optimized for computing the PLF on specific data types, e.g., a dedicated design for DNA data. Alternatively, one could also think of a system comprising several FPGAs that operate in parallel and comprise dedicated, optimized PRUs for specific data types.

5.5 4^{th} generation architecture

The work we present in this section represents the 4^{th} generation of our series of reconfigurable architectures for computing the PLF. Initially, we explored two alternative approaches (1^{st} generation in Section 5.2 and 2^{nd} generation in Section 5.3) for parallelizing the PLF on hardware. We found that a deeply pipelined tree-like placement of likelihood processing units (1^{st} generation design) can compute the PLF on fully balanced binary trees in a fast and memory-efficient way. However, a more flexible, vector-like arrangement of likelihood processing units (2^{nd} generation design) is better suited for real-world scenarios since it is completely independent of the tree-search strategy as well as the shape and size of the tree. After adopting this generic, vector-like arrangement of processing units for the PLF, we designed the 3^{rd} generation architecture (Section 5.4) that provided the full functionality for offloading all PLF functions from a real-world maximum likelihood program (RAxML) to a co-processor. The 3^{rd} generation design supports binary, DNA, protein, and RNA secondary structure data, as well as scaling

procedures to avoid numerical underflow, Newton-Raphson-based branch length optimization, and the calculation of transition probability matrices. We realized that such a highly flexible and generic architecture (capable of executing several PLF function types and handling data with a different number of states) can not be efficiently mapped (compared to performance on x86 architectures) onto present-day FPGAs.

The 4^{th} generation of the PLF architecture combines the concepts developed for the 2^{nd} and 3^{rd} generations, and it has been significantly improved with respect to resource utilization as well as performance. In addition, this last architecture implements a generic, FIFO-based interface to communicate with the outer world via, for instance, external memory or PCI Express. In the following, we describe the basic pipelined datapath of the PLF, the complete likelihood core as adapted for DNA data, the scaling unit, and the FIFO-based interface to the "outer" world.

5.5.1 Likelihood core

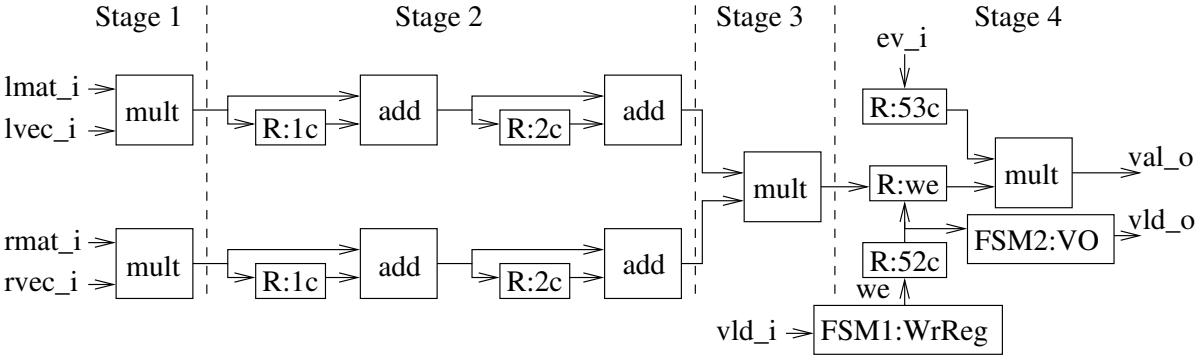


Figure 5.17: Block diagram of the pipelined PLF architecture for DNA data.

Figure 5.17 illustrates the pipelined datapath. There are four 64-bit input buses that are used to stream in the input data. We obtain the left and right transition probability matrices ($P(b_l)$ and $P(b_r)$, see Equation 2.9) over the *lmat_i* and *rmat_i* buses. The probability vector entries of the left and right probability vectors are streamed in ($\vec{L}^{(l)}$ and $\vec{L}^{(r)}$, see Equation 2.9) via the *lvec_i* and *rvec_i* buses. The pipeline is organized into four main stages which are distinguished by dashed lines in Figure 5.17. Stage 1 consists of two pipelined multipliers that operate in parallel and compute the product of the two input arguments from the left and right child nodes. Thereafter, Stage 2 accumulates the multiplier outputs within four clock cycles (remember that DNA data has four states). The results of the two parallel accumulation operations are combined in Stage 3 via a multiplication. The combination of stages 1, 2, and 3 implements Equation 2.9. Finally, Stage 4 executes the multiplication with the inverted eigenvector matrix (this is a numerical detail that is related to the exponentiation of the Q matrix as described in Section 2.4.2.1). Thus, a single pipeline instance can perform all operations required to multiply

a column of a transition probability matrix P with a probability vector entry $\vec{L}(i)$.

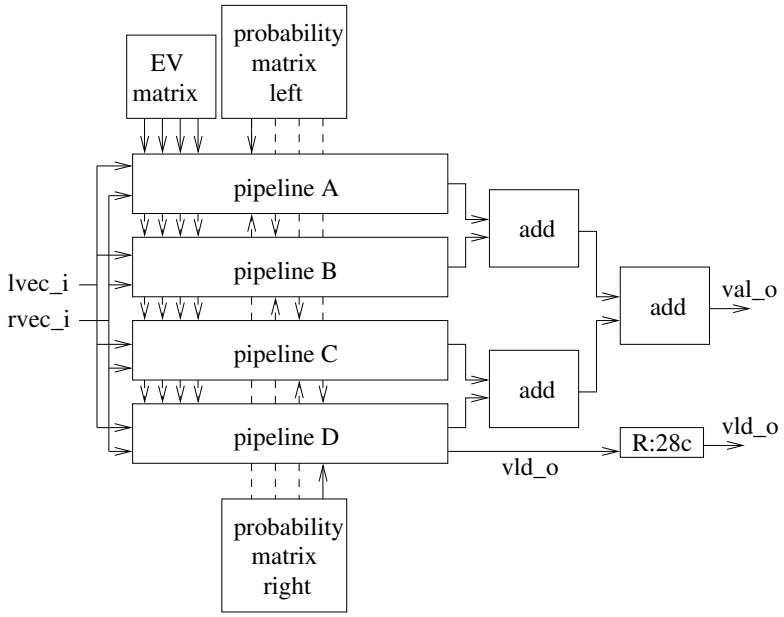


Figure 5.18: Block diagram of the likelihood core with 4 instances of the PLF pipeline.

The complete likelihood core for DNA data is shown in Figure 5.18. Four pipeline instances are required to accommodate the 4-state DNA model. Every pipeline instance operates on a different column of the two 4×4 probability transition matrices $P(b_l)$ and $P(b_r)$ that correspond to the branch lengths b_l and b_r of the branches that lead to the left and the right child nodes, respectively. All instances receive the same probability vector input and the same inverted eigenvector matrix. Note that only a single Q matrix is used for calculating the PLF on the entire tree. Thus, the eigenvector matrix for obtaining $P(b_l)$ and $P(b_r)$ remains constant. The results of the individual pipeline instances are then summed up using the adder tree depicted in Figure 5.18. The *val_o* output bus is used to stream out the resulting ancestral probability vector entry $\vec{L}^{(k)}$.

5.5.2 Scaling unit

As already mentioned, the output values (ancestral probability vector $\vec{L}^{(k)}$) of the likelihood core need to be checked for potential numerical underflow and scaled appropriately if required (for mathematical details refer to [183]). Therefore, we designed a pipelined probability vector entry scaling unit (Figure 5.19). The *scaler* unit works as follows: it monitors the values that are generated by the likelihood core on a per-site basis and scales/multiplies them by a constant factor if all values in an ancestral probability vector entry $\vec{L}^{(k)}(i)$ are smaller than a pre-defined threshold ϵ.

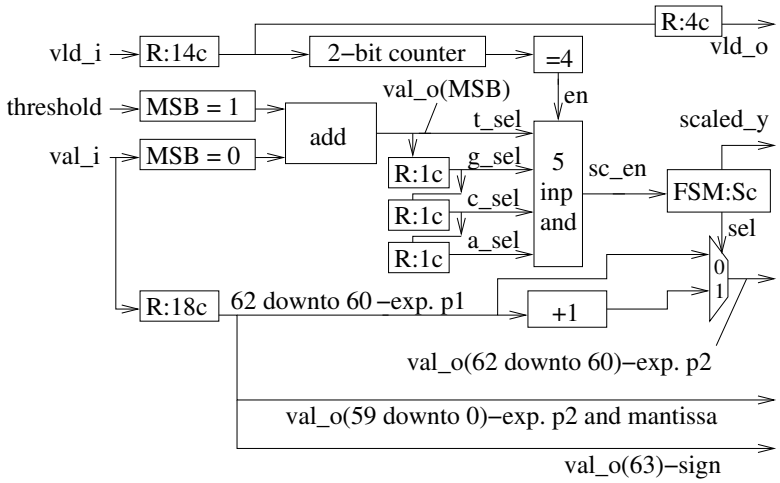


Figure 5.19: Architecture of the pipelined ancestral vector entry scaling unit.

Initially, an adder is used to compare the incoming probability values to ϵ. This is achieved by adding the negative value of the threshold to the input probabilities that need to be checked for scaling. The sign bit of the adder output indicates whether the values need to be scaled or not. Since we scale only when *all* four probabilities of an output vector are smaller than ϵ, we use three 1-bit shift registers with a delay of one clock cycle. Then, a five-input *and* operation is used to determine the value of the *sc_en* signal. Hence, *sc_en* is used to decide whether we need to scale or not. If we need to scale, we multiply all four probabilities in $\vec{L}^{(k)}(i)$ by 2^{256}. We implemented this multiplication via a 3-bit addition on the three most significant bits of the mantissa.

5.5.3 Architecture overview

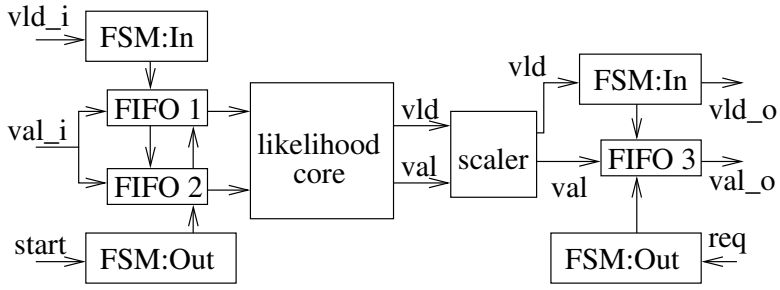


Figure 5.20: Top-level architecture of the PLF system.

Figure 5.20 illustrates how the likelihood core and the scaler unit are connected with each

other. The figure also shows how the core components are integrated with the FIFO input/output buffers that allow for efficient communication with the outer world (e.g., an external memory controller or a PCI Express bus). Incoming data are temporarily stored in the input FIFO buffers until enough data have arrived for computing an ancestral probability vector entry at a site i. These dedicated FIFO buffers, which are specifically adapted to the operation of our pipeline, allow for simplifying the pipeline architecture and thereby obtaining a more resource-efficient design with improved performance.

An important aspect of the proposed architecture is that it can easily be adapted to accommodate PLF computations on data with more states (e.g., protein data with 20 states or RNA secondary structure data models with 6, 7, or 16 states). The required modifications are straightforward since one only needs to appropriately adapt the adder/accumulator parts of the PLF pipeline, the likelihood core, and the number of shift registers in the scaler. More specifically, the current configuration of pipeline Stage 2 accumulates input values over four clock cycles. Extending this stage by more pairs of shift registers and adders allows for deploying the pipeline on data types with more states. A different number of pipeline stages is required in the likelihood core as well as for the adder-tree. The number of shift registers in the scaler component must be increased to match the number of states minus one. Also, the size of the *and* gate needs to be adapted.

5.5.4 Host-side management

In the following, we describe the software components required for using our FPGA-based system. To design an efficient hardware implementation, the entire "logic" that offloads *to* and steers operations *on* the hardware is retained on the PC. Thereby, we can deploy the hardware architecture to exclusively carry out the expensive PLF computations. Therefore, the host (PC) performs two main system tasks: i) manage memory and ii) initiate/steer computations on the FPGA.

5.5.4.1 Host-side memory management

The external memory (typically DRAM) on the FPGA board is entirely managed by the host. To this end, we developed an abstract interface that allows for conveniently managing vectors of fixed size. This interface organizes (keeps track of the organization of) the on-board external memory into an array of memory blocks of fixed size. The size of each memory block corresponds to the size of one full-length probability vector $\vec{L}$. Each individual memory block (probability vector) is associated with a unified vector address that contains a slot number and a bank number. The mapping of a unified address to a physical address in the external memory on the FPGA board depends on the implementation and the memory organization. Assuming a linear, byte-addressable memory space (which for example is used on the host to mirror the content of the external memory on the FPGA board), the physical address of a slot number Idx_{slot} is

$Addr_{phys} = Addr_{base} + Idx_{slot} * Size_{block}$, where $Addr_{base}$ is the start address of the memory region used for block storage and $Size_{block}$ is the size (number of bytes) of each block.

On the FPGA, the data organization is more complex than for a linear address space since one has to distribute each vector over multiple FPGA memory blocks to achieve high memory throughput. The host-side interface hides the intrinsic complexity of dividing the vectors into fragments and storing them in different FPGA memory blocks. The bank number, which is the second part of the unified vector address, is used to distribute data blocks among different external memory interfaces on the FPGA board. This allows for simultaneously accessing data blocks that reside in different banks. In the PLF context, reading and writing data in parallel through different memory interfaces is crucial to achieve maximum performance because the contents of two child vectors ($\vec{L}^{(l)}$, $\vec{L}^{(r)}$) can be read in parallel. Therefore, the data blocks associated with the two probability vectors of nodes l and r must be kept on different banks that are in turn associated with distinct memory interfaces. The host interface implements functions for data block transfer between the host DRAM and the FPGA. To achieve this in practice, we augmented the internal RAxML data structures by unified probability vector addresses.

5.5.4.2 Offloading operations

Apart from handling the entire memory management, the host also orchestrates the PLF operations that are executed on the FPGA. As mentioned above, the FPGA implements the PLF, which represents a key component of the RAxML tree-search algorithm. One basic component of the tree search is tree evaluation (computation of the overall log likelihood score of a tree topology), which requires to compute/update the ancestral probability vectors at the inner nodes of the tree via the Felsenstein's pruning algorithm. These updates of ancestral probability vectors, which are required when the tree topology is changed (in the course of the tree search), account for *the* computationally most intensive part of the PLF in RAxML (approximately 60%-70% of total runtime). Therefore, we entirely offload these computations to the FPGA.

To offload the computations, we need to transfer the constant probability vectors located at the tips (constant because the DNA sequences are known) to their corresponding memory locations in the on-board external memory. Thereafter, we traverse the tree (from the tips toward the virtual root) to calculate the ancestral probability vectors in post-order. To this end, we represent each post-order tree traversal by a traversal descriptor (an ordered list of tree node triplets (l, r, k)) which is initially transferred to the FPGA. More specifically, each element in the traversal descriptor contains a triplet of unified vector addresses (corresponding to the address of the parent node k and the two child probability vectors l and r) and a pair of branch lengths (b_l, b_r). The FPGA then executes the PLF as specified by the elements contained in the traversal descriptor *in sequence* until the PLF for all triplets in the list has been computed. For each traversal descriptor element, the reconfigurable architecture reads the data of the two child vectors and writes the result to the parent vector.

5.5.4.3 Host-side simulation

To test the memory-management concept, we simulated the memory-management interface and implemented the PLF in software on the host. We created a "mock-up" that implements the FPGA memory interface. Instead of transferring data between memories on the host and the FPGA, the mock-up emulates the FPGA memory contents and the target FPGA memory organization (using 2 banks consisting of 4 internal blocks each) on a x86 architecture.

Our experiments focused on transferring the probability vectors associated with the tips of the tree to the emulated FPGA memory and testing the iterations over the traversal descriptor (see Section 5.5.4.2). In our simulations, the PLF was computed using the FPGA emulator (running on the host), and the resulting probability vector was transferred back from the emulator via the interface to the associated host-side destination address in RAxML. Note that, for these simulations, the address spaces of RAxML and the SW emulation code for the FPGA were completely separated. After completing the iteration over the traversal descriptor, we compared the probability vector at the virtual root produced by the emulated FPGA to the corresponding vector calculated directly by RAxML. Thereby, we verified the correctness of the unified vector-address storage scheme in RAxML and tested the functionality of the memory allocation strategy.

5.5.5 System overview

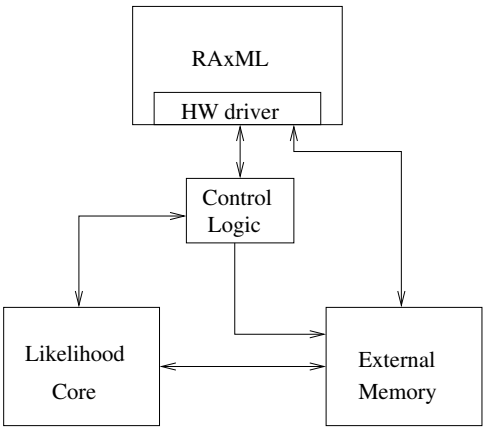


Figure 5.21: Conceptual design of the complete SW/HW system.

A schematic outline of the overall integrated system is provided in Figure 5.21. RAxML interacts with the reconfigurable hardware via the dedicated HW interface. On the FPGA side, a *Control Logic* module organizes all the incoming processing or data transfer requests that are received from the SW on the host side. The RAxML hardware interface can also directly access the external on-board memory to allow for overlapping data transfers with PLF com-

putations. The Control Logic module processes the received traversal descriptor and generates corresponding read/write addresses for the external memory on the FPGA board. The PLF calculations are carried out by the likelihood core and the ancestral state vectors are written back to external on-board memory. The Control Logic synchronizes the likelihood core calculations with the read/write operations to external memory. So far, we have implemented the basic RAxML interface as well as an initial testbench to generate and verify traversal descriptors in software. On the FPGA side, we have implemented, evaluated, and verified the likelihood core. The hardware description of the reconfigurable system is available for download at: `http://www.exelixis-lab.org/countLiCoGen4.php`.

5.5.6 Verification and performance

The reconfigurable architecture was implemented in VHDL and mapped to a Virtex 6 SX475T-2 FPGA. We verified the correctness of the hardware design by extensive post-place-and-route simulations using Modelsim 6.3f by Mentor Graphics. We generated input data for the simulations by using RAxML on real-world DNA sequences.

Table 5.4 shows the resource requirements for a single instance of the likelihood core on a Virtex 6 SX475-2 FPGA. A direct comparison to previous generations of our architecture can be misleading since each PLF hardware generation is organized in a different way and represents a distinct solution to the problem. Nonetheless, the basic computational pipeline that performs the matrix-vector multiplications of the PLF on DNA data (Figure 5.17) occupies approximately the same amount of resources as the respective pipelined 3^{rd} generation datapath (Figure 5.13). For this comparison, our 4^{th} generation PLF pipeline was mapped to the same FPGA used in the development of the 3^{rd} generation architecture (Virtex 5 SX95T-2). A significant difference between the two generations is that the 4^{th} generation pipeline is optimized for DNA data and can currently not accommodate statistical models for among-site rate heterogeneity (see [178] or [210]). The PLF pipeline has an initial latency of 115 clock cycles. Thereafter, it is able to compute one ancestral probability value in each clock cycle with a clock frequency of 293 MHz.

Resources	1 Core	8 Cores	Maximum
Slice Registers	28,385	226,977	595,200
Slice LUTs	19,596	159,263	297,600
Occupied Slices	7,823	55,692	74,400
DSP48Es	235	1880	2016

Table 5.4: Resource occupation and performance of the PLF architecture on a Virtex 6 SX475T-2 FPGA.

To evaluate the new PLF architecture, we compared it to the RAxML reference implementation on a state-of-the-art CPU (Intel i7 2600 at 3.4 GHz). Since this CPU provides 256-bit AVX vector instructions, we used the most efficient version of the PLF that deploys AVX vector intrin-

sics (available in RAxML-Light [184], `https://github.com/stamatak/RAxML-Light-1.0.5`). To conduct a fair comparison between the CPU and the FPGA, we introduced a new metric called VEUPS (Vector Entry Updates Per Second). A similar metric (CUPS; Cell Updates Per Second) is used in Bioinformatics to compare performance of pairwise alignment kernels among various hardware platforms [11, 121]. The vector-entry size depends on the number of states in the model. Therefore, VEUPS-based performance comparisons can be misleading if they do not refer to vector entries of the same size. We measured the VEUPS number of RAxML-Light on DNA data for a model that does not accommodate rate heterogeneity. Using one core of the Intel i7 CPU and a real-world DNA alignment, the peak CPU performance is 78.83 Mega VEUPS. The respective performance of a single FPGA likelihood core instance amounts to 73.54 Mega VEUPS.

This shows that the per-core VEUPS performance on a x86 architecture (when using AVX) is better than the respective per-PLF-core performance on a FPGA. Nonetheless, modern FPGAs allow for instantiating up to 8 such likelihood cores. However, such a dense design with several PLF hardware cores requires additional routing effort. Therefore, the maximum operating clock frequency decreases to 167 MHz. We estimate that the peak device performance that can be achieved for the PLF from the instantiation of 8 likelihood cores on a Virtex 6 FPGA amounts to 335.57 Mega VEUPS (approximately 42 Mega VEUPS per instantiated core).

5.6 Summary

This chapter presented four dedicated architectures for computing the PLF on reconfigurable logic. The first two—exploratory—architectures investigated alternative approaches for parallelizing PLF operations on FPGAs. The 1^{st} architecture is built around a tree-like placement of floating-point pipelines, achieving reduced memory requirements and speedups of up to 13-fold for computing likelihood scores on fully balanced binary tree topologies. The 2^{nd} generation employs a more flexible, vector-like placement of floating-point pipelines that operate independently of each other on different columns of the input alignment. We found that such a vector-like placement, despite yielding worse performance (speedups between 1.6 and 7.04), is more suitable for real-world applications since it is completely independent of the tree-search strategy as well as the topology and size of the tree.

The 3^{rd} generation architecture can be regarded as a proof-of-concept architecture for a potential ASIC PLF co-processor since it exhibits a comprehensive and versatile, yet static design. The architecture comprises hardware-optimized processing units that have the same functionality as the corresponding highly optimized likelihood functions in RAxML. As a consequence, the co-processor architecture can be used to conduct real-world phylogenetic analyses on morphological, DNA, RNA secondary structure, and protein data. Finally, the last (4^{th} generation) architecture presented in this chapter represents a DNA-optimized version of the main processing units of the previous generation. This 4^{th} generation architecture meets the computational

requirements of modern tree-search strategies and can easily be adapted to support data with more states (e.g., protein data).

For the evaluation of all FPGA architectures, significant effort was placed on establishing an as fair as possible performance comparison. We selected the most appropriate (based on the resource requirements of each design) FPGA to assess performance by comparing it to the respective fastest software implementation running on state-of-the-art CPUs at the time the experiments were conducted. A performance evaluation of the 3^{rd} architecture mapped on a Virtex 5 SX95T FPGA versus a SSE3-vectorized PLF implementation of RAxML running on an Intel CPU at 2.4 GHz revealed that FPGAs are not yet capable of efficiently accommodating a very generic implementation of the PLF. The FPGA co-processor is between 2.8 and 4.5 times slower than the corresponding software implementation. By deploying a significantly larger FPGA (Virtex 6 SX475T) and devising a DNA-optimized architecture (4^{th} generation), we found that FPGAs can be up to 4 times faster than state-of-the-art CPUs (Intel i7 CPU at 3.4 GHz running AVX-vectorized code).

For the 4^{th} and considerably more mature reconfigurable architecture, a novel approach for steering computations and managing on-board memory resources by maintaining a consistent view of the memory organization on the host was also presented. This approach, in combination with a dedicated control logic and the FIFO-based interface of the reconfigurable architecture, can be deployed by real-world applications such as RAxML to efficiently orchestrate the scarce and performance-critical on-board memory resources and offload operations to the FPGA architecture.

Chapter 6

Phylogeny-aware Short Read Alignment Acceleration

This chapter presents the acceleration of the phylogeny-aware short read alignment kernel of the PaPaRa algorithm [35] using FPGAs, vector intrinsics, and GPUs.

6.1 Introduction

As already mentioned, significant advances in molecular wet-lab sequencing techniques have led to a tremendous biological data flood in recent years. The term `short read` refers to DNA sequence data that are produced by next-generation sequencers. Current state-of-the-art pyrosequencing technologies can generate between 100,000 and 1,000,000 short reads. The read lengths typically vary between 30 and 450 nucleotides. To allow for an efficient and accurate phylogenetic analysis of such short read samples, novel maximum likelihood-based methods [65] have recently been introduced [35, 125, 192]. These approaches, known as phylogenetic placement algorithms, assign the short reads to a fixed, given reference phylogeny, that is, an unrooted phylogenetic (evolutionary) tree which is based on a given reference MSA. Before applying one of the aforementioned phylogenetic placement algorithms, all short reads must be aligned to the reference alignment.

This chapter focuses on the acceleration of the alignment kernel of PaPaRa [35] (PArsimony-based Phylogeny-Aware short Read Alignment) using FPGAs (Section 6.3), SSE4.1 SIMD intrinsics (Section 6.4), and GPUs (Section 6.5). The PaPaRa tool implements a novel method for aligning a—typically—large number of short sequence reads against a reference multiple sequence alignment (MSA) and a corresponding phylogenetic tree. HMMALIGN [53] can also be used to accomplish this task. With certain limitations, programs for *de novo* MSA such as MUSCLE [54] and MAFFT [102, 103] can also be deployed for this purpose. However, HMMALIGN, MUSCLE, and MAFFT align short sequence reads against a single, monolithic profile that is derived from the reference MSA without taking into account the corresponding

phylogeny. PaPaRa takes the phylogeny into account by calculating multiple profiles that are obtained from the phylogeny induced by the reference MSA. The short reads are aligned against each of these phylogeny-aware profiles and the best alignment for each short read is kept. Since a large number of pairwise alignments are computed (every query sequence (QS) is aligned against every edge of the reference tree (RT)), this operation dominates the runtimes of PaPaRa. The program is available as open-source code (`http://www.exelixis-lab.org/papara.tar.gz`). Profiling the PaPaRa source code revealed that the scoring function (the alignment kernel) accounts for 98% to 99.5% of overall execution time. Note that all pairwise alignment operations can be carried out independently. Hence, the algorithm exhibits a large degree of parallelism.

A characteristic property of PaPaRa and HMMALIGN is that they use dynamic programming algorithms for the alignment step. Dynamic programming alignment algorithms generally exhibit a time complexity of $O(mn)$ for aligning two sequences of length m and n against each other. This can become a limiting factor when either two long sequences or a large number of sequences are aligned. Therefore, optimization and acceleration efforts typically focus on optimizing these dynamic programming kernels. The underlying principle of PaPaRa is similar to the Smith-Waterman algorithm [175] with affine gap penalties [77]. However, the specific alignment kernel in PaPaRa is used to align a sequence (a short read) against an ancestral state vector that is derived from positions (branches) in the phylogenetic reference tree. Because of its generality and importance, optimization efforts have so far mainly been undertaken for the Smith-Waterman algorithm (SWA). Because of the analogies between the SWA and PaPaRa kernels, we briefly survey SWA optimization efforts.

Li *et al.* [117] presented a FPGA-based acceleration of the Smith-Waterman algorithm. They obtained a speedup of 160 compared to a C software implementation running on an Altera Nios II soft processor on the same FPGA (Altera Stratix EP1S40 FPGA). The reconfigurable hardware part, designed to accelerate dynamic programming matrix cell updates, had a maximum operating clock frequency of 3.1 MHz and a peak performance of 24.5 million CUPS (dynamic programming matrix cell updates per second). Yu *et al.* [214] presented an architecture for the Smith-Waterman algorithm based on a systolic cell array. Due to the efficiency of their design, they managed to instantiate 4,032 processing elements on a Xilinx XCV1000E-6 FPGA. Operating at a clock frequency of 202 MHz, their system achieved a performance of 814 billion CUPS.

Several alternative implementations [84, 96] for accelerating the Smith-Waterman algorithm using FPGAs exist. However, because the PaPaRa alignment kernel differs from the standard Smith-Waterman implementation, we omit a more detailed review of FPGA-based accelerators at this point. Note that there also exists related work on accelerating the (more complex) dynamic programming kernel of HMMER [53]. For an overview of FPGA accelerator architectures for the Viterbi algorithm used in HMMER refer to [61]. Performance results vary between 0.7 and 20 million CUPS.

There exists extensive literature on vectorizing SWA with SIMD instructions on general pur-

pose CPUs: [22] for the Intel i860, [206] for the Sun Ultra Sparc, and [62, 163, 164] for Intel x86 CPUs. The above implementations deploy fundamentally different techniques for vectorizing the algorithm: [62, 164, 206] deploy SIMD instructions to speed up the pairwise alignment of two sequences at a fine-grain level. They exploit the data parallelism in the calculations of a single dynamic programming matrix (typically denoted as intra-task parallelism). However, the inherent wavefront parallelism limits the parallel efficiency of these approaches, which motivated the introduction of increasingly sophisticated methods to alleviate these limitations. Other implementations use a fundamentally different approach. They do not vectorize individual pairwise alignment computations. Instead, they simultaneously compute multiple pairwise sequence alignments. In [22], a 64-bit special purpose register is divided into four parts for simultaneously aligning a single sequence against four other sequences. This represents a straightforward application of data parallelism, generally referred to as inter-task parallelism. In other words, the basic alignment algorithm is executed sequentially but is applied simultaneously to multiple data. The authors obtained a 6-fold speedup over the sequential implementation. Recently, Rognes introduced SWIPE [163], a highly optimized inter-task vectorization approach for modern x86 architectures that uses SSE instructions and achieves a speedup of up to 6 over the fastest intra-task SSE vectorization [62].

Also, several approaches have already been assessed for accelerating the SWA on GPUs. Initial efforts used OpenGL [202]. Later implementations, after the introduction of CUDA, led to the development of SW-CUDA [124], CUDASW++ [120], and CUDASW++2.0 [121]. According to [163], the most efficient CPU and GPU implementations yield comparable performance on current state-of-the-art hardware (SWIPE on a typical quad-core CPU and CUDASW++2.0 on a NVIDIA GeForce GTX 480).

6.2 PArsimony-based Phylogeny-Aware short Read Alignment (PaPaRa)

In a phylogenetic tree, known sequences of living species (extant taxa) are located at the tips of the tree. The inner nodes represent hypothetical common ancestors of these species. The actual sequences of the ancestors are unknown. Hence, different methods for accommodating the uncertainty of ancestral states have been introduced in the context of scoring criteria for phylogenetic trees.

In PaPaRa, ancestral states are obtained via maximum parsimony (MP). Based on a fixed, given reference MSA (denoted as RA), the key idea is to find *the* phylogenetic tree which explains the data (MSA) by the least number of mutations (see Sections 2.4.1 and 4.2). For DNA data, every edge (branch) b of the reference tree (RT) can be represented by a parsimony state vector $A_b = A_b^1, ..., A_b^n$, where each A_b^i represents the parsimony state of the RA at site i (i.e., column i). Each entry A_b^i is encoded as a bit vector. Each bit corresponds to one of the four DNA characters. This representation differs from the simpler case of pairwise sequence alignment

(e.g., SWA), where each element of both input sequences represents *exactly* one character and not potential, alternative character states. In PaPaRa, an individual A_b^i entry can either be an `A`, `C`, `G`, or `T`, or any combination thereof (e.g., `A` and `T`). This representation is used to encode the uncertainty of an ancestral sequence state. Thus, in the case of DNA data, using these bit vectors is analogous to ambiguous character representations (e.g., `M` representing either `G` or `T`). The bit vectors can also be used for other input data types such as protein data.

As mentioned before, the QS are aligned against all ancestral states derived from the edges of the RT. At each edge, an additional node (a 'virtual root') is inserted, for which an ancestral state vector is computed. When the ancestral state vector has been calculated, the virtual root is removed again and inserted into another edge. In conjunction with this ancestral state vector, PaPaRa uses an additional signal which provides information about the gap (indel) distribution in the RT. For this purpose, we use a supplementary flag (CGAP) at each site i. This flag is used to appropriately adapt (calibrate) the scoring function of the dynamic programming algorithm to the indel pattern as encoded in the RT and RA. The CGAP signal is calculated along with each ancestral state vector at each edge based on the rules described in [35]. The scoring function of the dynamic programming algorithm is provided in Equation 6.1. Note that the default gap and mismatch penalties are different from the default values reported in [35]. Equation 6.1 recursively defines the score of the dynamic programming matrix cell $S^{i,j}$ in column i and row j for aligning site A_b^i of the ancestral state vector against site B^j in the QS.

$$
\begin{aligned}
CG^i &= \begin{cases} 3 & \text{if CGAP is set for site } i \\ 0 & \text{otherwise} \end{cases} \\
(GP^i_{OE}, GP^i_E) &= \begin{cases} (4,1) & \text{if} CG^i = 0 \\ (0,0) & \text{otherwise} \end{cases} \\
M^{i,j} &= \begin{cases} 0 & \text{if } A^i \text{and} B^j \text{match} \\ 3 & \text{otherwise} \end{cases} \\
I^{i,j} &= S^{i,j-1} + 3 \\
D^{i,j} &= min \begin{cases} S^{i-1,j} + GP^i_{OE} \\ D^{i-1,j} + GP^i_E \end{cases} \\
S^{i,j} &= min \begin{cases} S^{i-1,j-1} + M^{i,j} + CG^i \\ D^{i,j} \\ I^{i,j} \end{cases}
\end{aligned}
\tag{6.1}
$$

For instance, an ancestral state A^i at site i with $A^i = [1100]$ means that As and Cs in the QS under consideration can be matched against alignment site i of the ancestral state without incurring a mismatch penalty. Thus, for scoring mismatches via the respective function $M^{i,j}$,

the default mismatch penalty of 3 will be used, unless the bit corresponding to the character at position j in the QS is set in A^i. When this matching condition is met, the score returned by $M^{i,j}$ is 0. The special way in which match and mismatch penalties are treated along with the phylogeny-aware adaptive scoring scheme (CG^i) in PaPaRa represent a key difference to standard dynamic programming methods for sequence alignment.

The PaPaRa algorithm consists of two phases. Initially, all QS are aligned against all ancestral state vectors in the RT. For each QS, only the best alignment score is retained. Thus, given a RT with r taxa, m sites, and q QS, the program needs to calculate $O(rq)$ alignments performing $O(rqm^2)$ operations. During the initial phase, the actual alignments are not generated (i.e., the trace-back step is not performed) for performance reasons. During the second phase (given the best scores for all QS), the actual alignments are generated by aligning each QS again, but only against the respective best-scoring ancestral state vector found for the QS during the initial alignment step. As already mentioned, the initial (all query sequences against all ancestral probability vectors) alignments account for more than 98% of overall runtime. Thus, this initial alignment phase represents the natural candidate for acceleration.

6.3 Reconfigurable system

The underlying idea of our design is to offload the scoring function (alignment kernel) calculations to dedicated hardware components (denoted as Score Processing Units or SPUs) on the FPGA, while the software implementation on the PC side is used to orchestrate computations, collect the scores, and perform the final trace-back step of the alignment algorithm. The software implementation and the hardware description of the reconfigurable architecture are available as open-source code at: `http://www.exelixis-lab.org/FPGA_papara.tar.gz`. In the following, we present the reconfigurable architecture of the Score Processing Unit.

6.3.1 Score Processing Unit (SPU)

Figure 6.1 depicts the block diagram of the top-level design. The control of the SPU is orchestrated by four FSMs that are located in the top left corner of Figure 6.1. The FSMs operate in a master-worker scheme; the master FSM (*M_FSM*) initiates and synchronizes the operation of the three worker FSMs (*Q_FSM*, *R_FSM*, and *PR_FSM*).

The *Q_FSM* is triggered when a QS is received. The main purpose of the *Q_FSM* is to generate the write-enable signal as well as the correct write addresses for the *Q_MEM* memory with a capacity of 200x16 bits. In each of the 16 memory lines, a total of 100 QS nucleotides are stored. The *Q_MEM* memory is used to store a single, incoming QS until an ancestral reference state vector (RS) arrives. When the QS is stored in memory, the *Q_FSM* sends a signal to the *M_FSM* which then switches to reference-awaiting state.

Similar to the *Q_FSM*, the *R_FSM* is responsible for storing the RS in the RS FIFO buffer (*REF_FIFO_BUF*). The main difference between the *R_FSM* and the *Q_FSM* is that once the

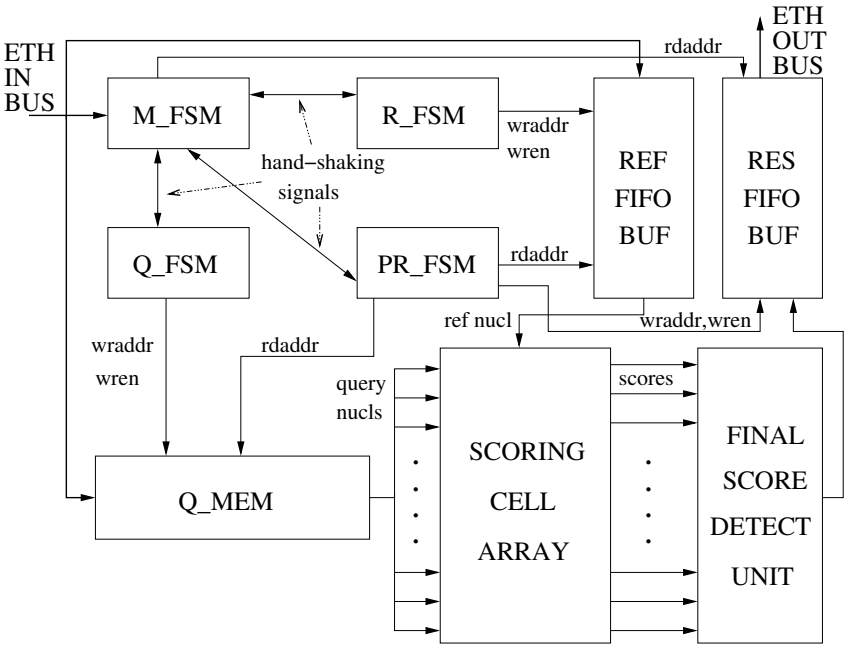


Figure 6.1: Top-level block diagram of the Score Processing Unit (SPU).

first position of the RS FIFO buffer has been filled, that is, the first state (A^0) for the first site of the RS has arrived, the *R_FSM* immediately notifies the master FSM.

The *M_FSM* interprets the notification from the *R_FSM* as a signal to start the actual computations, and therefore triggers the processing FSM (*PR_FSM*). The *PR_FSM* generates read addresses for the *Q_MEM* and *REF_FIFO_BUF* memories as well as all the required signals for the operation of the *SCORING_CELL_ARRAY*.

The *SCORING_CELL_ARRAY* represents the computational kernel of the *SPU* and consists of 100 scoring cells (*SCs*) that operate in parallel. Figure 6.2 illustrates the architecture of the *SC*. The output ports of each *SC* are connected to a neighboring *SC* as well as to the *FINAL_SCORE_DET_UN* unit (see Figure 6.1). The *FINAL_SCORE_DET_UN* unit selects the output of the *SC* that corresponds to the last line of the dynamic programming matrix and also determines the minimum value in that last line. When the entire matrix has been processed, the *FINAL_SCORE_DET_UN* unit writes the minimum value to the *RES_FIFO_BUF* output FIFO buffer.

To accommodate scoring requests for query sequences whose length exceeds the fixed number of *SCs* available in the *SPU*, we integrated a loop control module. Figure 6.3 provides an abstract representation of this subsystem. Based on the current position (index) of the nucleotides in the query sequence that are provided as input at each clock cycle, the *Loop_Control* module either feeds the output of the last *SC* (100th in the present implementation) or a constant value to the input ports of the first *SC*. The constant value is set to zero and can be regarded as the

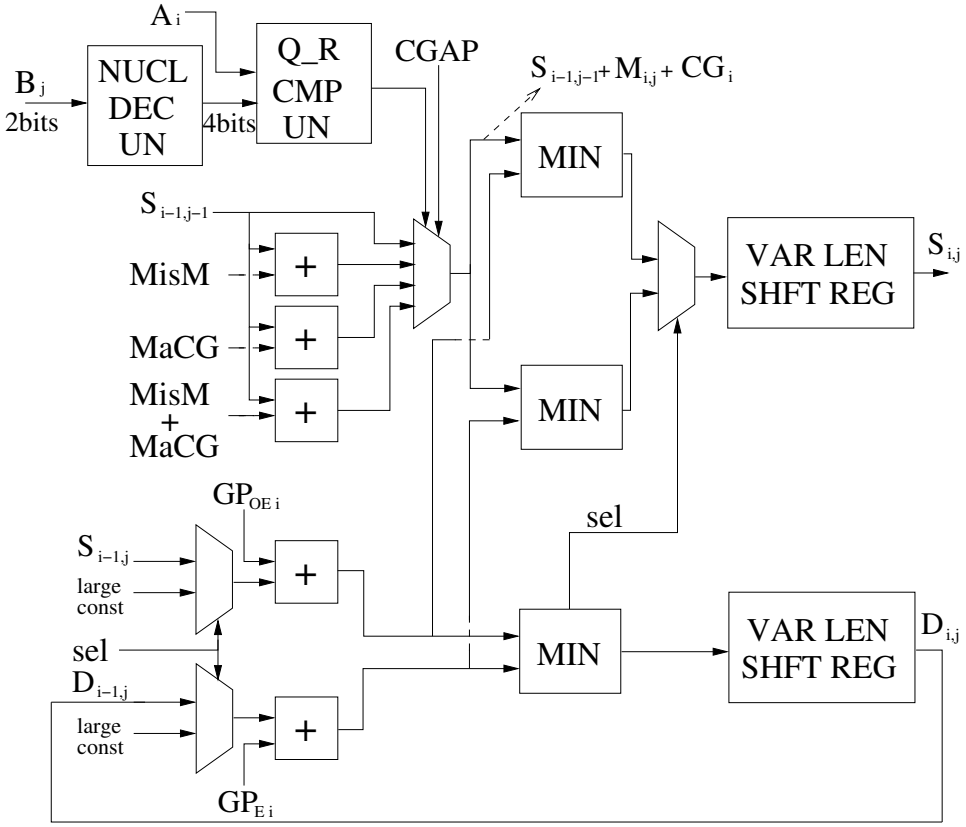


Figure 6.2: The scoring cell (*SC*) architecture.

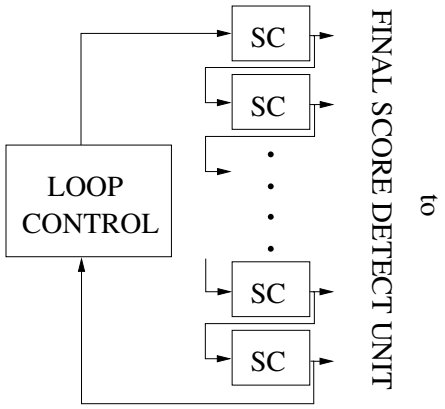


Figure 6.3: Loop control mechanism for accommodating query sequence lengths that are larger than the number of available *SCs*.

row of the dynamic programming matrix at position -1. This constant zero value essentially represents the dynamic programming matrix row that immediately precedes the next row to be calculated.

The *SC* unit (Figure 6.2) calculates Equation 6.1 (see Section 6.2). Initially, the B_j value, which represents a 2-bit coding of a nucleotide in the QS, is transformed into the corresponding 4-bit representation by the *NUC_DEC_UN* unit. Then, the *Q_R_CMP_UN* unit performs a comparison between the QS nucleotide and the current RS position. Based on the result of this comparison *and* the CGAP signal, the 4-to-1 multiplexer selects one out of the four possible values for the intermediate $S^{i-1,j-1} + M^{i,j} + CG^i$ value. The four possible values are either the output of a neighboring SC (value $S^{i-1,j-1}$) or one of the following three constant values: i) MisM (mismatch, 3), ii) MaCG (match with a CGAP, 3), and iii) MisM + MaCG (mismatch with a CGAP, 6).

The three parallel components denoted as *MIN* in Figure 6.2 are used to select the minimum value among the input signals (see Equation 6.1 in Section 6.2). In fact, only two *MIN* comparison components are required. The additional *MIN* module is used to shorten the critical path and thereby obtain a higher operating clock frequency.

Finally, the *VAR_LEN_SHFT_REG* components are variable-length, RAM-based shift registers that increase the latency of the *SCs* relative to the QS length. The latency of the computational part of the SC is 1 clock cycle. Since the *SCORING_CELL_ARRAY* comprises 100 *SCs*, scoring requests with a QS length of less than 100 nucleotides do not require additional latency because the processing array can operate on a different site during every clock cycle. For instance, when a matrix for a QS length of 350 nucleotides is computed, the *VAR_LEN_SHFT_REG* registers will increase the latency to 4 clock cycles. The cell values are temporarily buffered in the pipeline stages of the *SCs* until the *SCORING_CELL_ARRAY* finishes the operations at some site i and is able to proceed to site $i + 1$. During the 4 clock cycles, the QS is provided as input to the *SC* array in blocks of 100 nucleotides per cycle. For 350 nucleotides, the last 50 *SCs* of the array will receive undefined input data in every 4th clock cycle and the respective outputs will be ignored by the *FINAL_SCORE_DETECT_UNIT*.

6.3.2 Implementation and verification

The SPU architecture was implemented in VHDL and mapped on a Virtex 5 SX95T-2 FPGA. Extensive post-place-and-route simulations were conducted to verify the functionality and correctness of the proposed architecture. As simulation tool we used Modelsim 6.3f by Mentor Graphics. We used the PaPaRa source code to generate appropriate testbenches for real-world biological datasets.

Thereafter, the HTG-V5-PCIE development board with the same Virtex 5 SX95T FPGA was used for testing on the actual chip. The main objective of these tests was to verify the correctness of the SPU architecture. We used an advanced verification tool (the Chipscope Pro Analyzer) to monitor the input and output ports of the SPU.

6.3.3 System overview

After successful verification of the SPU architecture (post-place-and-route simulations *and* tests on an actual FPGA), we integrated a simplified communication protocol between the FPGA board (HTG-V5-PCIE) and the host PC. For this purpose, a Dell Latitude E4300 series laptop with an Intel Core2 Duo P9400 processor (2.4 GHz, Ubuntu) was used. The communication between the FPGA board and the PC was established over Gigabit Ethernet using a dedicated UDP/IP core for direct PC-FPGA communication [10]. Figure 6.4 illustrates the complete system.

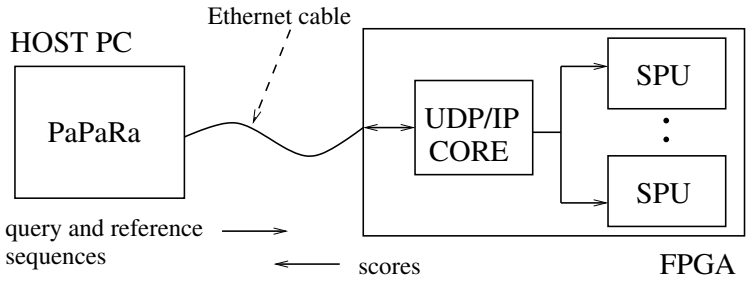


Figure 6.4: The FPGA-based acceleration system.

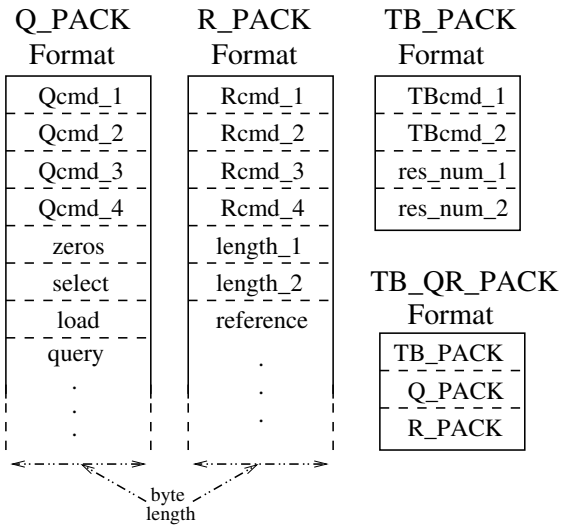


Figure 6.5: Basic packet formats.

A special sequence of packets needs to be transmitted such that the PaPaRa software (on the PC) can trigger SPU operations. An alignment request requires the transmission of a packet that contains the query sequence (*Q_PACK*) followed by a packet that contains the reference sequence (*R_PACK*) against which the query sequence shall be aligned. Every X packets,

where X is the size of the *RES_FIFO_BUF*, a *TB_PACK* packet (stands for Transmit Back) is transmitted. The *TB_PACK* packet indicates that the software is ready to receive the results from the previous alignment requests that are stored in *RES_FIFO_BUF*. Figure 6.5 illustrates the supported packet formats.

To further optimize the communication process, we configured the FPGA to transmit and receive JUMBO frames (i.e., IP packets that are longer than the standard minimum length of 1,500 bytes). This allowed us to transmit only a single Ethernet packet for each alignment request. Thus, a common alignment request now requires the transmission of a new packet type, the *QR_PACK*. Here, the *Q_PACK* can be concatenated with the *R_PACK* into the same large UDP packet. Also, a modified transmit-back request *TB_QR_PACK*, asking for the results of a previous alignment request, can be inserted at the beginning of the common alignment request *QR_PACK* format to replace the stand-alone *TB_PACK* packet.

Each packet format starts with a unique command code. In the query and reference packets, the command code occupies four bytes while in the transmit-back packet it occupies two bytes. In addition, the *Q_PACK* contains one byte with zeros which is ignored by the SPU, a selection code which is used by the *FINAL_SCORE_DETECT_UNIT* to multiplex the results of the 100 parallel *SCs*, a load code which is used to control the variable length shift registers of the *SCs*, and finally the query sequence. Accordingly, the *R_PACK* contains a 2-byte field that contains the reference sequence length and the raw reference sequence data. Finally, in the *TB_PACK* format, the 2-byte TB command is followed by another 2-byte field that contains the total number of results that shall be transmitted back. Note that the number of results can not exceed the size of the *RES_FIFO_BUF* memory.

We created an experimental extension of the PaPaRa program on the PC side. The original code is used for reading input files, encoding QS and RS into the appropriate format, sending SPU-compliant UDP packets to the FPGA, receiving the result packets from the FPGA, performing the trace-back step, and generating the actual QS alignments for the best insertion edges. The program extension was created in C++ using the Boost library (`www.boost.org`) for sending and receiving UDP packets.

The fast response time of the FPGA led to a difficult technical challenge for the PC application when synchronous communication is used. After the JUMBO UDP packet containing the QS and RS has been sent (QR_PACK) to the FPGA, the FPGA sends back the resulting scores within a very short amount of time (only 6 cycles after the end-of-frame signal of the UDP packet is set). Because program execution on the PC is blocked via a system call until the whole UDP packet has been sent, the answering packet may already have been returned by the FPGA before the PC can actually start receiving it, which once again requires a system call. Incoming UDP packets are generally not buffered, which means that if no corresponding receive operation has been initiated, the packets returned by the FPGA can be lost. Therefore, we deploy an asynchronous communication mechanism that relies on a dedicated thread which initiates a receive operation *before* the QS and RS are sent to the FPGA. We used the

asynchronous I/O operations as provided by the Boost library to implement this functionality.

Note that the place-and-route static timing report revealed that the maximum operating clock frequency of an SPU on a Virtex 5 SX95T FPGA is 101.52 MHz. In order to avoid additional synchronization problems with the software and at the same time verify the functionality of the proposed communication protocol, the same clock signal of 125 MHz was used for both the communication components on the FPGA and the SPU. By communication components we refer to the TEMAC (Tri-Mode Ethernet Media Access Controller) and the UDP/IP core. This SPU over-clocking caused the incorrect computation of a fraction of the scores. According to the error report that was generated by the software during the final alignment phase, the fraction of incorrect scores due to over-clocking the prototype system ranged between 10% and 20% of the overall SPU score requests.

6.3.4 Performance

Here, we present a performance assessment for the accelerator architecture when multiple SPUs are instantiated on a large Virtex 6 FPGA. All results presented in the current section refer to post-place-and-route Xilinx reports (after the implementation process).

Resources	1 SPU V6	12 SPUs V6	1 SPU SYS V5
Slice Regs	7,725	92,248	8,366
Slice LUTs	29,026	347,344	29,103
Occup. Slices	7,791	88,036	9,059
BlockRAMs (36k)	5	60	5
BlockRAMs (18k)	5	60	10
TEMACs	-	-	1

Table 6.1: Occupied resources on the V5SX95T and V6HX565T devices.

Table 6.1 provides resource usage reports for three hardware configurations. The first column of Table 6.1 provides the resources occupied by a single SPU instance on a Virtex 6 HX565T-2 FPGA (`1_SPU_V6`). According to the static timing report, the maximum operating frequency for this unit is 140.92 MHz. The 12_SPU_V6 architecture comprises 12 independent SPU instances with a maximum operating frequency of 111.17 MHz. Finally, the `1_SPU_V5_SYS` implementation, which was described in the previous section, contains 1 SPU instance, a TEMAC, and a UDP/IP core.

To compare performance, we executed the software-only implementation of PaPaRa on an Intel Core i5 750 CPU running at 3.2 GHz. Initially, we measured the total execution time for the scoring phase (aligning all QS against all ancestral states) required by the PaPaRa implementation for 4 real-world biological datasets. The PaPaRa software contains an algorithmic optimization (the so-called 'early stopping criterion') which decreases the total number of ma-

trix cells that need to be calculated. This trick improves the runtime of the PaPaRa software by a factor of 2 to 3 [35]. The current FPGA design does not implement this optimization since this would require a non-trivial re-design of the pipeline datapath for calculating the matrix entries *and* of the communication protocol.

	Alignment kernel			Trace	Kernel speedup VS		Application speedup VS	
Dataset	PC (base)	PC (opt)	FPGA	back	PC (base)	PC (opt)	PC (base)	PC (opt)
D218_200	1,359	841	1.82	9.6	748.8	463.7	119.9	74.6
D218_500	2,868	1,890	4.01	19.9	715.2	471.2	120.8	79.9
D500_200	4,125	1,772	6.4	13	641.3	275.5	213.0	91.9
D500_500	7,872	3,784	14.18	22.9	555.1	266.8	212.9	102.7
D855_200	12,516	4,269	19.37	23.6	646.2	220.4	291.8	100.0
D855_500	23,947	9,604	42.69	41.6	560.8	224.9	284.6	114.4
D1604_200	38,333	12,109	60.50	42.5	633.6	200.1	372.6	118.0
D1604_500	69,815	22,684	133.26	68.8	523.9	170.2	345.9	112.6

Table 6.2: Total execution times (in seconds) of the alignment kernel and the trace-back step of the PaPaRa algorithm, as well as the respective speedups of the FPGA system for the alignment kernel and for the complete application. The standard algorithm implementation is denoted as *base*, while the optimized version is denoted as *opt*.

Nonetheless, to conduct a fair comparison, we present performance data for the standard software implementation as well as for the optimized software implementation with the early stopping criterion. For a given dataset, the standard software implementation performs exactly the same number of cell updates as the current hardware design. The second phase of PaPaRa ('alignment phase') is always executed on the PC for the software-based FPGA-accelerated implementation. Table 6.2 provides execution times and the speedups that can be achieved by offloading the alignment kernel to a Virtex 6 FPGA with 12 SPUs. The input dataset names in the left column denote the number of taxa in the original biological dataset followed by the average QS length (see [35]).

The required data transfer rate for a SPU-based accelerator system is given by the following formula:

$$I = [(SPU\_N * SC\_N * Q\_CD) + R\_CD]/CLK\_P,$$

where `SPU_N` is the number of SPUs in the design (12 in our implementation), `SC_N` is the number of scoring cells in the processing array of each SPU (100 in our implementation), `Q_CD` is the number of bits required to represent a QS character (2 in our implementation), `R_CD` is the number of bits required to represent a RS ancestral state (5 in our implementation), and finally `CLK_P` is the clock period. Based on the above formula, for the given configuration, data has to be provided to the SPUs at a rate of 31.2 GB/s to achieve the maximum possible speedups reported in Table 6.2. This transfer rate is higher than what we can currently achieve with existing PC-FPGA communication methods like Gigabit Ethernet (max. 125MB/s) or PCI Express (max. 16 GB/s), but the requirement can be met by using Block-RAM-based input/output

buffers for storing input data. Ideally (i.e., if the input data fits into the buffer), each query and reference sequence will have to be transferred to the respective buffer only once and can then be fed into the SPUs several times. Clearly, the efficiency of this approach depends on an appropriate input/output buffer size and requires appropriate buffer management. Note that the communication overhead for transmitting the results (final matrix scores) produced by the SPUs back to the PC can be neglected. In contrast to input data transfers, every SPU only generates a 16-bit alignment score every INTERVAL(Q_CD) * R_CD clock cycles. The INTERVAL (Query length) function returns the number of clock cycles spent by the *SCORING_CELL_ARRAY* for each column/site of the matrix (each ancestral vector state).

6.4 SIMD implementation

6.4.1 Inter-reference memory organization

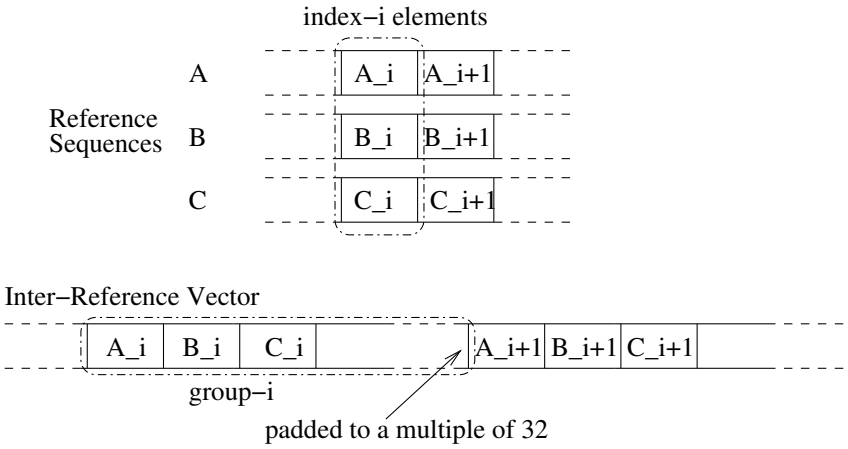


Figure 6.6: Example of the inter-reference memory organization on SIMD and SIMT platforms. All index-i elements are grouped together in group i. Groups are padded to a multiple of 32 for performance reasons.

An inter-reference memory organization model, similar to the one described in [163], is deployed for both the SIMD as well as the SIMT implementations. On the SIMD (x86) architecture the inter-reference organization facilitates the vectorization of the code, while on the SIMT architecture it allows for coalesced global memory accesses and thereby high device throughput. In Figure 6.6, we illustrate this generic inter-reference memory organization approach. A certain number W (a work-group) of reference sequences (RS) is organized in one large array, the 'inter-reference vector', which consists of consecutive groups of elements. Each group contains all R elements for one specific position/index of the RS. The elements of a group are placed contiguously and all groups are placed sequentially in memory. In other words, the group containing the $i+1$th elements follows the group containing the ith elements. The group

size (number of elements per work-group) varies for the SIMD and SIMT architectures. A similar organization is used to store the dynamic programming matrix D (see Equation 6.1): each entry $D^{i,j}$ consists of a group of W elements, and the groups in $D^{i,j}$ and $D^{i+1,j}$ occupy consecutive memory locations. The group width W of the SIMD implementation depends on the selected integer data type and the x86 target architecture (see Section 6.4.2 for details). For SIMT architectures, group sizes are multiples of 32 unsigned integers to ensure that the global memory is accessed in chunks of 128 bytes. Moreover, all memory transactions are aligned automatically (every address is a multiple of 128). The actual number of reference elements that a group contains is either 32 (number of unsigned integers in the group) or larger, depending on the compression of the reference entries (number of reference entries that are stored in an unsigned integer, see Section 6.5.4).

6.4.2 Vector intrinsics

The $R * Q$ independent alignment operations of the PaPaRa scoring phase can easily be distributed to multiple cores using threads. This corresponds to a MIMD (Multiple Instruction, Multiple Data) parallelization scheme. This MIMD parallelization scales linearly with the number of cores. Further performance improvements can be obtained by exploiting the capability of modern CPUs to simultaneously work on multiple data elements by means of dedicated SIMD (Single Instruction, Multiple Data) instructions. The SIMD implementation of the algorithm is straightforward. Based on the input data organization described in Section 6.4.1, individual instructions of the sequential implementation (i.e., the naïve implementation of Equation 6.1) can be transformed into corresponding SIMD instructions. In contrast to the sequential implementation, which aligns a single QS against a single ancestral reference at a time, the SIMD implementation simultaneously aligns a single QS against W ancestral reference sequences on each x86 core.

While the number of instructions that are necessary to calculate individual entries in the dynamic programming matrix is similar for the sequential and the SIMD implementations, the throughput is increased by W since each SIMD instruction works on W times more data compared to its sequential counterpart. It is therefore crucial to reduce the amount of memory used by the alignment algorithm such that frequently accessed data are kept in cache. During the scoring phase (dynamic programming matrix computations), we do not keep the entire dynamic programming matrix in memory but only a single line. This is possible because the calculation of a matrix entry $D^{i,j}$ in row j only depends on other values of D that have previously been calculated either in the same row or in the preceding row $j-1$ (see Equation 6.1).

To make the SIMD implementation more generic, we hide the specifics of the actual SIMD instruction set (e.g., SSE) in a thin abstraction layer called a generalized vector unit, which is implemented using C++ templates. The generalized vector unit can be instantiated with different data types (16-bit/32-bit integers) and vector unit widths (8 or 4 units for 16-bit or 32-bit integers, respectively, when using SSE). The top-level algorithm uses abstract vector data

types (integer vector of 8x16 bits) and vector operations (e.g., load data from a memory location into a 8x16-bit integer vector, add two 8x16-bit integer vectors, etc.). Here, the group width W corresponds to the width of the vector unit (i.e., either 4 or 8 for 16-bit or 32-bit integer vectors) on our primary target platform, an Intel x86 CPU with SSE instructions. This ensures that the element groups in the RS and the dynamic programming matrix D can be directly loaded into vector registers. The generalized vector unit for Intel x86 CPUs was mapped to SSE4.1 intrinsics. Note that SSE version 4.1 is only required for using 32-bit integers. For 16-bit integers SSE version 2.0 or higher is sufficient. We have verified that the concept of the generalized vector unit also works correctly for other vector instruction sets (e.g., ARM NEON or Intel AVX instructions).

6.5 SIMT implementation

Despite the fact that the SIMT version of the algorithm was tested on NVIDIA hardware, we chose to use OpenCL instead of CUDA. While CUDA can potentially yield improved performance since it is better adapted to the specifics of NVIDIA hardware, OpenCL allows for using the same code on different GPUs (e.g., AMD/ATI). Moreover, OpenCL can also be used for general-purpose multi-core systems, which is particularly convenient for testing and debugging.

6.5.1 Inter-task parallelization

On the SIMT platform, each alignment kernel invocation calculates one dynamic programming matrix. To efficiently execute the kernel on a GPU, every individual dynamic programming matrix calculation is assigned to a distinct thread (inter-task parallelization). An alternative parallelization scheme using intra-task parallelization would assign each task (dynamic programming matrix calculation) to a thread block, and all threads within that block could then cooperate to accomplish the task. Liu *et al.* [120] investigated both the inter-task as well as the intra-task parallelization approaches for porting the SWA to SIMT platforms. They found that inter-task outperforms intra-task parallelization. However, the intra-task approach requires significantly less device memory per thread. The intra-task approach is also appealing in cases where a large number of independent pairwise alignments need to be performed. However, in PaPaRa, each QS is aligned against a large number of RS. This alignment workload can be grouped into many 1-to-W alignments, which naturally fits the inter-task parallelization scheme. Previous experiments with the SWA by other authors showed that if the problem at hand is such that the inter-task approach can be deployed, it consistently outperforms the intra-task approach on SIMD [163] *and* SIMT [120] architectures. The inter-task approach is more communication-efficient since frequent synchronization between threads operating on a single, shared dynamic programming matrix is not required. Threads only need to be synchronized once upon kernel termination. Therefore, we did not further investigate the intra-task approach for PaPaRa. Figure 6.7 depicts the inter-task parallelization approach we use here.

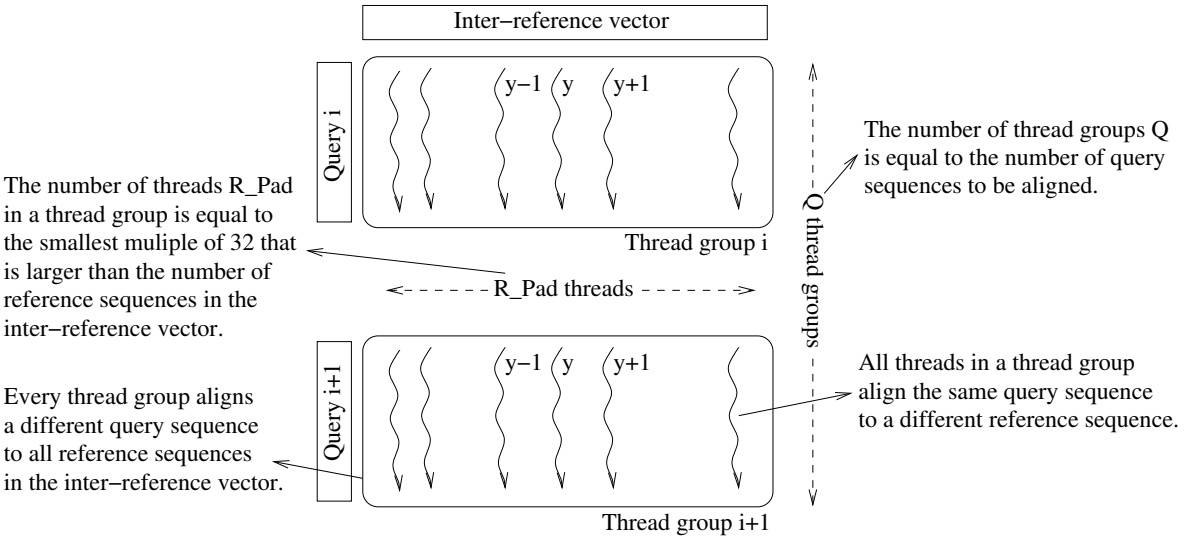


Figure 6.7: Example of the inter-task parallelization strategy. Thread y of group i aligns QS i to RS y. The number of threads in a thread group depends on the number of RS in the inter-reference vector. The number of thread groups depends on the number of QS. Every thread group aligns a different QS to all RS, and all threads in a thread group align the same QS to all RS.

6.5.2 Block-based matrix calculation

A straightforward approach to calculate the PaPaRa dynamic programming matrix is to compute one row after the other in a diagonal direction as shown in Figure 6.8. Since only the last matrix row is required for calculating the next row, the memory requirements of this approach depend on the size of the inter-reference vector. However, this approach requires off-chip global memory and does not allow for using on-chip shared memory. The main reason for this is that the size of the inter-reference vector in real-world scenarios will typically exceed the amount of shared memory available on a representative SIMT platform. Furthermore, the entire row needs to be calculated before proceeding to the next row. To alleviate these shortcomings and to be able to use shared memory, we devised a block-based approach. The matrix cells are calculated in square or rectangular blocks of adjustable size. The block size is a function of the length of the query sequences, the amount of available shared memory, and the number of RS. Figure 6.9 depicts this blocked matrix calculation model.

Due to the diagonal direction of the PaPaRa matrix calculations and the limited amount of on-chip memory, it is not possible to use shared memory efficiently along the entire inter-reference vector. This means that the very first and very last parts of it need to be calculated using global memory to store the intermediate values of the row fraction they are operating on. The part of the inter-reference vector residing in the rectangular blocks can, however, be calculated using shared memory. Using rectangular blocks at either end of the RS requires the

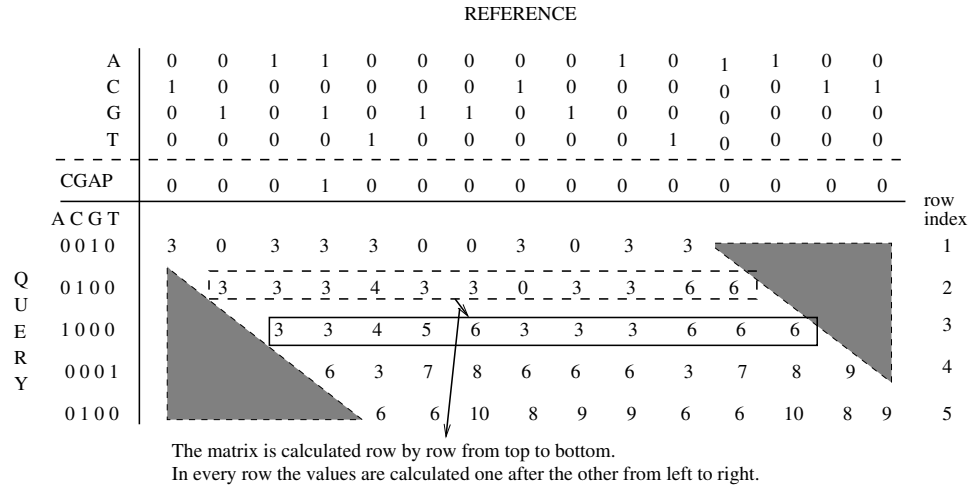


Figure 6.8: The straightforward approach for calculating the scoring matrix consists in calculating one row after the other. In every row the values are calculated from left to right. Therefore, an entire row is calculated before proceeding to the next one.

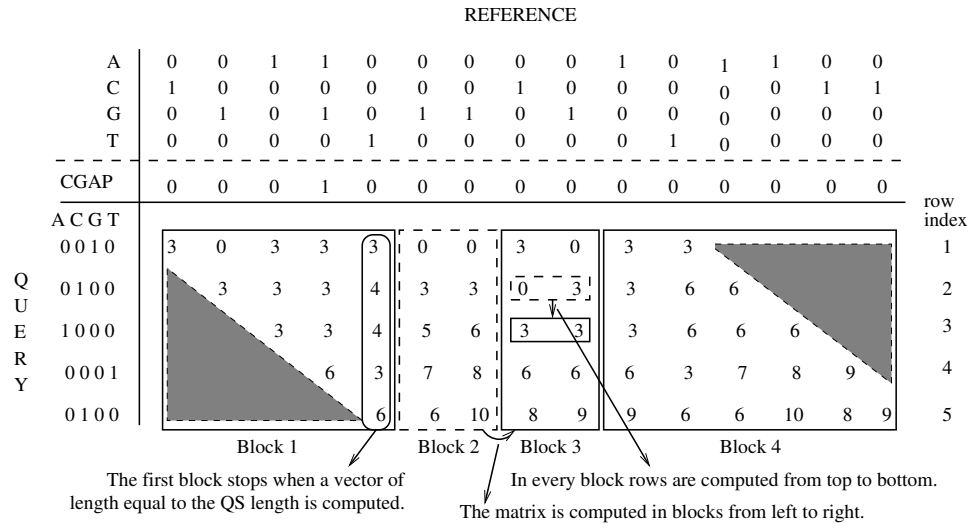


Figure 6.9: Block-based calculation of the PaPaRa scoring matrix. The dynamic programming matrix is calculated in blocks starting from the left-most square block and proceeds toward the right-most square block. In every block the rows are calculated one after the other. Block 1 stops when the last row of the matrix is reached. It operates on global memory due to its size. The main part of the reference sequence is processed in rectangular blocks (Blocks 2 and 3). These blocks are of significantly smaller size and operate on shared memory. Block 4 processes the last part of the reference sequence in global memory. Blocks 1 and 4, which operate on global memory, represent an engineering trade-off to avoid *if-else* conditional statements in the GPU kernel that would slow down Blocks 2 and 3. To process real-world reference sequences, many more than two iterations over the rectangular blocks are required.

evaluation of additional conditional (*if-else*) statements in the kernel code. Note that evaluating conditionals can substantially deteriorate GPU performance. Thus, we use global memory at either end as a trade-off to circumvent this problem and to avoid evaluating conditionals that would slow down processing of the main part of the inter-reference vector. Our approach only requires a fixed amount of global memory, irrespective of the length of the RS (stored in the inter-reference vector), since the full length RS is processed by iterating over blocks of fixed size. In the example provided in Figure 6.9, Blocks 1 and 4 use global memory, while Blocks 2 and 3 operate on shared memory.

6.5.3 Loop unrolling

An advantage of the blocked approach over the straightforward approach is that the amount of global memory required for the computation of a single dynamic programming matrix does not depend on the length of the RS. Nonetheless, code complexity increases since three nested for-loops are required to compute the entire matrix: i) one loop iterates over the rectangular blocks, ii) the second loop iterates over the query sequence, and iii) the innermost loop iterates over the reference fraction in the current block. We observed that the anticipated performance gain by using shared memory was reduced by the increased code complexity, which hindered the GPU threads to efficiently execute the blocked version of the OpenCL kernel. To solve this problem, we unrolled the innermost for-loop that iterates over the fraction of the reference corresponding to the rectangular blocks.

Loop unrolling significantly improved kernel performance but comes at a cost: the number of RS that the kernel can align is hard-coded. However, this limitation is not problematic since, in typical real-world scenarios, users will align thousands of reference and query sequences. In this case, the amount of dynamic programming matrices to be computed is organized in groups that contain a fixed number of RS. Thus, the OpenCL kernel can be launched several times on those groups. We opted for using a fixed number of RS based upon an in-depth investigation of the impact of the shared memory size setting on OpenCL kernel performance. In our implementation, we use 15 KBs of shared memory because a significant slow-down was observed when larger amounts of shared memory were used. The number of RS per kernel launch is fixed (hard-coded) to 320, which allows for unrolling the innermost loop 12 times. Thus, we were able to completely remove this loop because the selected shared memory size only allows for processing 12 columns of the dynamic programming matrix in each block.

6.5.4 Data compression

The global memory access pattern is also performance-critical on SIMT platforms. Therefore, we compress the part of the inter-reference vector that is processed by the blocks in shared memory to reduce the frequency of global memory accesses. Every element (inter-reference vector entry) in the inter-reference vector requires 5 bits (4 bits for the parsimony state plus 1

bit for the phylogenetic gap signal CGAP), which allows the use of one unsigned 32-bit integer value to store six elements. This boils down to only two global memory accesses for every 12 elements (or two global accesses per rectangular block). Since square blocks are not unrolled, the inter-reference vector entries in such blocks are also not compressed for the sake of code simplicity, and to allow for the efficient execution of threads. Figure 6.10 outlines how the inter-reference vector is organized on the SIMT platform. The uncompressed initial and final parts of the vector require one integer per element, while the compressed part requires one integer per six elements. Note that all three parts of the vector (uncompressed initial and final parts and the compressed intermediate part) are padded to a multiple of 32 unsigned integers.

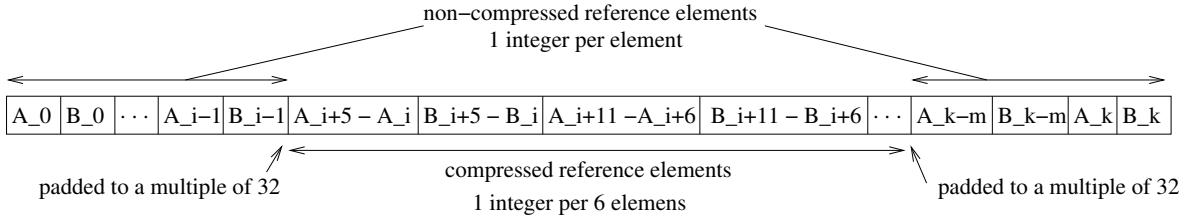


Figure 6.10: Compressed inter-reference memory organization. The non-compressed fractions of the vector are processed by the square blocks, while the compressed part is processed by the rectangular blocks.

6.5.5 OpenCL application

The complete OpenCL application consists of tasks performed by the CPU (host program) and operations that are offloaded to the GPU (kernel functions). The host program handles the inter-reference memory organization and compresses the input RS. Once the references have been rearranged (organized into the inter-reference vector) and compressed, they are stored in pinned (non-pageable) contiguous memory space. The query sequences are also stored in contiguous blocks, each the length of the longest query sequence. Allocating pinned memory for the query and inter-reference vectors allows for fast CPU-to-GPU data transfers via PCI Express. After the data (query and reference sequences) have been transferred to the global GPU memory, the host program launches a one-dimensional kernel of size $Q * R\_Padded$, where Q is the number of query sequences and R_Padded the smallest multiple of 32 that is greater than the number R of RS. Those $Q * R\_Padded$ threads are organized in Q local groups of size R_Padded. This local, group-based thread organization simplifies the indexing of the query sequences in the kernel function. Note that, due to memory limitations, the number Q of query sequences that is aligned per kernel call usually only represents a fraction of the total amount of query sequences in the input dataset.

Finally, the kernel function implements the actual PaPaRa alignment kernel. The local group index in the kernel indicates the QS while the thread index indicates the RS. Thus all threads

in a local group align the same query sequence to all reference sequences in device memory. The query and reference sequences are retrieved from global memory and the intermediate values (the values of the dynamic programming matrices) are either stored in global memory (first and last square blocks) or in shared memory (rectangular blocks). As described before, all global memory accesses are correctly aligned (to 128 bytes) to minimize the performance impact of high-latency global memory accesses.

6.5.6 Performance

To assess performance of the OpenCL SIMT implementation, we used a heterogeneous system equipped with an Intel i7 2600 CPU running at 3.4 GHz (SIMD platform) and a NVIDIA GeForce 560 GPU with 336 CUDA cores and 1 GB DDR5 device memory (SIMT platform). We measured total execution times as well as GCUPS (Giga Cell Updates Per Second) for kernel executions on real-world datasets.

Reference length	Execution times			GCUPS			Speedup GPU vs	
	Seq	SSE(4)	GPU	Seq	SSE(4)	GPU	Seq	SSE(4)
500	40.79	1.11	1.89	0.49	18.01	8.4	21.58	0.59
1000	90.45	2.58	2.5	0.44	15.52	14.4	36.18	1.03
5000	494.65	14.30	9.23	0.40	14.30	21.2	53.59	1.55
10000	1006.1	29.27	18.31	0.40	13.66	21.6	54.95	1.60
50000	5103.4	319.92	90.95	0.39	6.25	21.9	56.11	3.52
100000	10369	1785.8	181.31	0.38	2.24	22.0	57.19	9.85
500000	51448	9005.3	906.21	0.39	2.22	22.1	56.77	9.94

Table 6.3: OpenCL kernel performance for RS with different lengths. The table shows execution times (in seconds) to align 1,250 QS to 320 RS, as well as GCUPS performance and speedups.

Initially, we focus on the performance of the OpenCL kernel for different input dataset sizes. To this end, we investigate the behavior of single kernel launches with different RS lengths between 500 and 500,000 nucleotides. Every kernel launch aligns 1,250 query sequences with an average length of 100 nucleotides to 320 RS. As shown in Table 6.3, the performance of the OpenCL implementation improves with the RS length until the peak performance of 22 GCUPS is reached. For very short RS (e.g., 500 nucleotides), kernel performance drops below 50% of peak performance. As already stated, the block-based implementation uses global memory for the first and last square blocks of the matrix and shared memory for the intermediate rectangular blocks. When the RS is short, global memory will predominantly be used for dynamic programming matrix calculations since the square blocks almost cover the entire matrix. In other words, because of the short reference length, the potential performance gains by using shared memory are negligible since the majority of operations is conducted on global memory. For longer sequences however, the majority of operations is carried out on shared memory.

Therefore, improved GPU performance is attained.

SIMD performance behaves differently with increasing RS length. The cumulative performance on four cores is highest for the short references with 500 base pairs (18.01 GCUPS) and decreases linearly to reference lengths of 10,000 base pairs (13.66 GCUPS). We observed a further substantial performance deterioration on very long reference sequences (e.g., 6.25 and 2.22 GCUPS for lengths of 50,000 and 500,000, respectively). This is due to the increased number of cache misses when longer references are analyzed. Our SIMD implementation keeps one line of the dynamic programming matrix in memory. Each matrix entry corresponds to a vector of 8x16-bit values (16 bytes). For reference lengths of 10,000, the matrix line requires roughly 160 KB, which fits into the L2 cache (256 KBs per core) of the Intel i7 2600 CPU. For a reference length of 50,000, the matrix row occupies 800 KBs and does not fit into the L2 cache any more. Therefore, most matrix cell calculations need to access the slower L3 cache (8 MBs shared by all cores). The slowdown is even larger for the longest sequences under examination since the data do not fit into L3 cache either. Currently, we do not expect these slowdowns to be problematic for real-world scenarios because current wet-lab sequencing devices rarely produce reads exceeding a length of 1,000 base pairs [99]. However, when future sequencing technologies (e.g., PacBio (`www.pacificbiosciences.com`)) are capable of generating a large number of reads longer than 1,000 base pairs, a blocking technique similar to the one devised for the SIMT implementation could also be applied to the SIMD implementation.

Number of queries	Execution times			GCUPS			Speedup GPU vs	
	Seq	SSE(4)	GPU	Seq	SSE(4)	GPU	Seq	SSE(4)
100	39.60	1.14	0.78	0.40	14.10	20.2	50.8	1.46
250	98.31	2.78	1.87	0.41	14.27	20.9	52.6	1.48
500	197.57	5.57	3.7	0.40	14.33	21.1	53.4	1.51
750	296.23	8.40	5.5	0.40	14.24	21.1	53.9	1.53
1000	395.59	11.20	7.4	0.40	14.28	21.2	53.4	1.51
1250	494.65	14.30	9.23	0.40	14.29	21.2	53.6	1.52

Table 6.4: OpenCL kernel performance for different number of query sequences. The table shows execution times (in seconds) to align the QS to 320 RS with length 5,000, as well as GCUPS performance and speedups.

We also investigated the performance of the OpenCL kernel for different numbers of query sequences. We chose a fixed RS length of 5,000, since this represents an average case scenario, and this is close to our GPU's peak performance levels. Keeping the reference length constant, we launched the kernel multiple times using different query sequence numbers. The results of this performance assessment are provided in Table 6.4. While kernel performance is practically independent of the number of query sequences, there is a slight performance improvement up to 500 QS for the SIMD implementation. This is caused by the initial overhead required for transforming the RS into their inter-reference representation, which has to be done once for

each block of W RS but is independent of the QS number. For more than 500 QS, this initial overhead becomes negligible. As expected, the performance of the sequential implementation is completely independent of the QS number. We did not conduct any tests to examine the performance of the kernel for different average query sequence lengths since the PaPaRa algorithm has been designed for aligning short read QS (e.g., Illumina or 454 reads) to a RA.

We developed and optimized the OpenCL kernel mainly for the NVIDIA Fermi GPU architecture. Nonetheless, we also executed some exploratory tests on an AMD/ATI GPU (a RADEON HD 6970 with a theoretical peak performance of 2,703 GFLOPS). As expected, the OpenCL kernel could be executed on the ATI system without any substantial modifications, but we observed poor performance. On a subset of the real-world dataset used to evaluate performance of the hybrid CPU-GPU system (see Section 6.6.2), we measured a performance of 13.2 GCUPS (the NVIDIA GTX 560 GPU with 1,075 GFLOPS peak performance delivered 18.9 GCUPS on this dataset). One would expect a 3 to 4 times better performance for a fully ATI-tuned kernel. However, note that, in contrast to the NVIDIA Fermi architecture, the AMD/ATI architecture heavily relies on Instruction Level Parallelism (ILP). Thus, to attain peak performance, it may be necessary to rewrite the alignment kernel such that the OpenCL compiler can group similar instructions more efficiently in an SIMD-like manner. Thus, while OpenCL code is portable, achieving satisfying performance still requires architecture-aware tuning.

6.6 Hybrid CPU-GPU approach

6.6.1 System overview

To achieve the maximum possible performance for a typical CPU-GPU system, we also designed a hybrid PaPaRa version that simultaneously uses all available cores as well as a GPU. In addition to multi-threading, available in the original implementation [35], the CPU part of the hybrid system uses the SIMD implementation described earlier with a vector/group-width $W = 8$ and 16-bit scores. Initially, the algorithm generates and groups the RS into 'blocks' of W sequences. These blocks are stored in a work queue and concurrently retrieved one by one from the worker threads which align all QS against the W reference sequences in the block. Once all queued blocks have been calculated, the master thread resumes control and for each QS only recomputes the alignment for the respective best-scoring QS/RS pair to perform the backtracking step.

The GPU part of the hybrid system extends this mechanism by an additional GPU thread. In analogy to the CPU threads, the GPU thread consumes blocks from the same work queue. A key difference is that, while each CPU thread only removes one block from the queue at a time (i.e., W matches the native vector width of the CPU), the GPU works on a higher number of RS at a time (e.g., 320). Thus, the GPU thread can remove up to 40 blocks from the work queue at a time. The GPU thread continues to obtain and work on multiple blocks until the block queue is empty. Thereby, the CPU cores and the GPU compete for RS. For a sufficiently

large number of RS, this yields good load balance between the CPU cores and the GPU. This approach can also be extended to use more than one GPU.

The GPU DRAM size limits the number and the length of QS that can be transferred to the GPU at each invocation of the alignment kernel. For long QS the number of QS per invocation has to be small enough such that there is sufficient DRAM available, while for short QS the number has to be high enough to achieve good performance (aligning a small number of short QS greatly reduces the performance of the GPU aligner). The hybrid CPU-GPU algorithm dynamically optimizes the number of QS based on the available amount of DRAM and the actual QS lengths. Thereby, we ensure that each kernel invocation operates on the largest possible QS number. This dynamic load balancing allows for stable performance over different dataset shapes with different sequence length distributions (see Section 6.6.2).

Currently, the OpenCL implementation faces a technical difficulty on NVIDIA GPUs. Once a kernel is invoked, an internal thread of the GPU driver executes a busy-wait, apparently (the driver is closed-source) waiting for the GPU to finish. If a `clFinish` call is deployed on the CPU to wait for completion of the GPU kernel, this will, in addition to the internal driver thread, cause the calling thread to execute a busy-wait, effectively wasting the computational power of two CPU cores. While CUDA allows to select between a busy-wait (for fine-grain, fast kernels) and a lazy-wait, this option does not yet exist in OpenCL. From the values shown in Table 6.4, we can roughly estimate that the speedup of the GPU over two CPU cores is approximately three-fold in the best case (assuming that 14 GCUPS are obtained on four cores). This means that wasting two CPU cores for interacting with the GPU has a negative impact on overall system performance. An analysis of the NVIDIA OpenCL library showed that the internal busy-wait implementation executes `sched_yield` function calls which should yield CPU cycle time to other threads. In practice, this call has no positive effect and overall system performance decreases proportionally to the threads that are executing a busy-wait. This busy-wait issue is a well-known Linux problem (see `http://www.mail-archive.com/linux-kernel@vger.kernel.org/msg91605.html`). To temporarily circumvent this issue, we use a customized shared library to replace (using `LD_PRELOAD`) all `sched_yield` calls by `usleep` calls. This actually yields CPU cycles to other threads at the cost of increased kernel call latency, but only of the order of a few milliseconds. This latency increase is not critical here because for sufficiently large numbers of QS each GPU kernel invocation requires a few seconds to complete (see Section 6.5.6). Using this work-around we can leverage the entire computational power of the GPU and *all* CPU cores.

6.6.2 Performance

We assessed overall performance of the hybrid CPU-GPU algorithm using two representative real-world datasets. The experiments were performed on the same system as described above (Intel i7 2600 CPU, NVIDIA GeForce GTX 560 GPU) using all four CPU cores. The first test dataset (1604.PRANK) has already been used to evaluate the original implementation of

PaPaRa in [35]. This dataset contains 802 RS of length 3,060 and 16,040 QS with a mean length of 100 base pairs. We did not use other datasets from the original study because the hybrid CPU-GPU algorithm targets larger datasets. The second dataset (16S.B.ALL) is from a recent study comparing PaPaRa to a newly developed algorithm [130]. The unoptimized, proof-of-concept implementation of PaPaRa performs better than competing alignment approaches on the latter real-world dataset at the cost of substantially higher runtimes. This dataset consists of 13,822 RS of length 6,857 and 13,820 QS of lengths that vary between 29 and 483.

Dataset	$T_{scoring}$(s)	T_{all}(s)	$GCUPS_{CPU}$	$GCUPS_{GPU}$	$GCUPS_{all}$
1604.PRANK	227.21	273.12	13.38	20.04	33.42
16S.B.ALL	12943.9	13111.4	13.87	20.00	33.87

Table 6.5: System performance of the hybrid CPU-GPU algorithm.

Table 6.5 shows the performance of the hybrid CPU-GPU algorithm on the two datasets. Column $T_{scoring}$ provides the runtime for the scoring phase. Column T_{all} shows the overall runtime for the whole algorithm, including the pre-processing of input files and the generation of the actual alignments. These pre- and post-processing steps are unoptimized sequential tasks that are performed on the CPU. The overall CPU performance (accumulated performance on four CPU cores) is shown in column $GCUPS_{CPU}$. Overall GPU performance is provided in column $GCUPS_{GPU}$. On both datasets, the relative contributions of the CPU cores and the GPU are very similar; the CPU and the GPU contribute 40% and 60% of the overall GCUPS to the accumulated CPU-GPU system performance, which is indicated in the last column ($GCUPS_{all}$). All GCUPS values refer to sustained GCUPS since they include the overhead induced by load imbalance between the CPU and the GPU. Load imbalance is observed when either one of the CPU threads or the GPU finish last and require the other computational resources to wait. The sustained real-world performance of the hybrid system corresponds, almost exactly, to the performance we obtained for the previous benchmarks (Section 6.5.6).

6.7 Summary

This chapter initially presented a hardware/software implementation for boosting performance of a novel short read alignment method that simultaneously aligns reads to reference MSAs and corresponding phylogenetic trees. The reconfigurable architecture was verified on a Virtex 5 FPGA and the functionality of the communication protocol was tested using Gigabit Ethernet. The hardware architecture achieved speedups for the score calculation phase of PaPaRa ranging between 170 and 471 on a Xilinx Virtex 6 FPGA compared to the sequential version of PaPaRa on an Intel Core i5 CPU running at 3.2 GHz. To the best of our knowledge, this represents the first FPGA-based accelerator architecture for this novel alignment kernel.

Thereafter, the PaPaRa kernel was implemented in OpenCL. Several informed design decisions and optimization techniques were applied to achieve the best possible performance. The inter-reference memory organization allowed for the efficient use of global memory, while the block-based approach was devised to exploit on-chip shared memory. The blocked implementation allowed for loop unrolling and data compression, which further improved performance. Apart from the thoroughly optimized OpenCL implementation, we also developed a vectorized version of the kernel using SSE4.1 intrinsics to investigate how state-of-the-art SIMD platforms perform for PaPaRa. On an Intel i7 2600 CPU (using all four physical cores), we obtained a peak performance of 18 GCUPS and an average performance of 12.3 GCUPS. A mid-range gaming GPU like the GTX 560 delivered peak and average performance of 22.1 and 18.4 GCUPS.

Finally, we combined and coupled the SIMD and SIMT implementations to create a hybrid CPU-GPU system that can exploit all available computational resources in a representative modern desktop system (GTX 560 GPU and i7 2600 CPU). The total system peak performance was 33.8 GCUPS. By efficiently exploiting all computational resources, we were able to fundamentally improve the applicability of the highly accurate PaPaRa algorithm to large real-world datasets.

We deployed the "competing programmer approach" to provide an *as fair as possible* performance comparison between the competing architectures. The results in this chapter corroborate the tradition of FPGAs outperforming (approximately one order of magnitude) CPUs and GPUs for the implementation of alignment kernels, as well as the observations made in [163] that the computational capabilities of modern CPUs and GPUs are in the same order of magnitude provided that highly optimized alignment kernels are developed on both platforms.

Chapter 7

Tree Searches on Phylogenomic Alignments with Missing Data

This chapter presents a generally applicable mechanism for reducing memory footprints and number of floating-point arithmetic operations of likelihood-based phylogenomic analyses in proportion to the amount of missing data in the alignment. Furthermore, a set of algorithmic rules to efficiently conduct tree searches via subtree pruning and regrafting moves using this mechanism is introduced.

7.1 Introduction

In this chapter, we study the time- and memory-efficient execution of SPR moves for conducting tree searches on gappy phylogenomic multi-gene alignments (also known as super-matrices) under the maximum likelihood (ML [65]) model by example of RAxML [182]. While we use RAxML to prove our concept, the mechanisms presented here can easily be integrated into all Bayesian and ML-based programs that conduct tree searches and are hence predominantly limited by the time and space efficiency of likelihood computations on trees.

Sanderson *et al.* [170] recently described a solution to the problem of tree reconstruction with missing data based on "terraces" of trees with identical quality. Initial work by Stamatakis and Ott [189] on methods for efficiently computing the likelihood on phylogenomic alignments with missing data focused on computing the likelihood and optimizing branch lengths on a single, fixed tree topology using pointer meshes. Here, we address the conceptually more difficult extension of this approach to likelihood model parameter optimization (for parameters other than branch lengths) and tree searches which entail dynamically changing trees. We describe and make available as open-source code a generally applicable framework to efficiently compute the likelihood on dynamically changing tree topologies during SPR-based tree searches. SPR moves represent the most widely used tree-search technique in state-of-the-art programs for phylogenetic inference. In addition, we implement full ML model parameter optimization under

the proposed mechanism and take advantage of the memory footprint reduction potential that was only mentioned as a theoretical possibility (without providing an actual implementation) in [189].

7.2 Underlying concept

The underlying idea of our mechanism to accelerate likelihood computations is that we do not need to conduct computations nor allocate memory for data that is not present in gappy phylogenomic alignments. Typically, a large fraction (50%-90%) of phylogenomic alignments consist of undetermined characters that are used to denote the complete absence of sequence data for specific gene/taxon pairs. Given two genes, G_0 and G_1, that comprise a total of n taxa, a sequence for both genes will typically not be available for every taxon; for some taxa, molecular data will only be available for G_0 and for others only for G_1. The missing per-gene sequences for each taxon are usually just filled up with undetermined characters. Figure 7.1 provides an example of such a gappy multi-gene dataset with some missing sequence data in each gene. Given the way undetermined characters are modeled in most present-day phylogenetic likelihood function implementations, we can observe that adding a taxon which consists entirely of gaps to a tree at an arbitrary branch will not change its likelihood. This method is used in all popular likelihood-based programs such as GARLI [4], PHYML [80], MrBayes [165], and PhyloBayes [112].

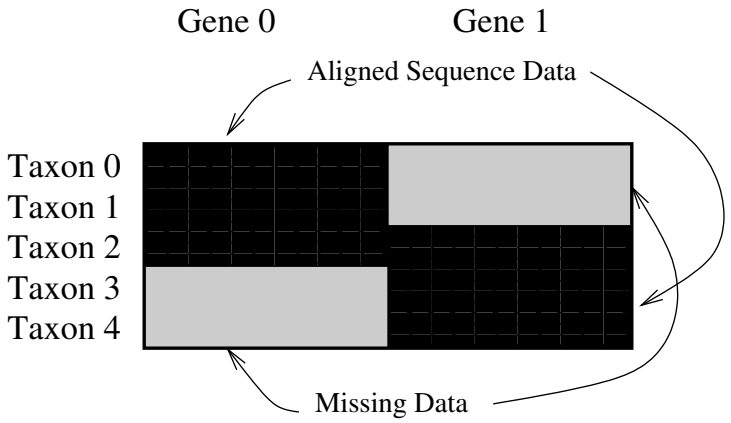


Figure 7.1: A gappy phylogenomic multi-gene alignment with a gappyness of 40% (40% of the data are missing).

If one conducts a partitioned analysis of the multi-gene dataset as outlined in Figure 7.1, and one applies a per-partition (per-gene) estimate of branch lengths, we observe the following: for a given tree t that comprises all 5 taxa, we may compute the overall likelihood as $LnL = LnL(t|G_0) + LnL(t|G_1)$, where $LnL(t|G_i)$ is the likelihood of the tree t for gene G_i *restricted*

to the taxa for which sequence data are available in gene i. This means that, for the example dataset in Figure 7.1, we only need to compute and add the likelihoods of two 3-taxon trees instead of two 5-taxon trees for genes G_0 and G_1 under the standard likelihood implementation. Restricting the global tree t to a per-gene subtree for the available molecular data therefore allows to save both a significant amount of floating-point operations as well as a significant amount of memory for storing ancestral probability vectors. The memory footprint reduction that can be achieved is roughly proportional to the gappyness of the respective alignment.

While the above idea is per se simple, the key challenge consists in designing rapid methods to extract the subtrees induced by genes from the comprehensive tree t, and to maintain the pointer meshes that are used to keep track of the per-gene tree topologies (see Figure 7.2) in a consistent state. Moreover, conducting tree searches, using for instance SPR moves, requires a set of rules to dynamically update the pointer meshes when a specific SPR move induces a change in a individual per-gene subtree. Because of the high complexity of this approach, a correct implementation of SPR moves represents an algorithmic as well as a software engineering challenge.

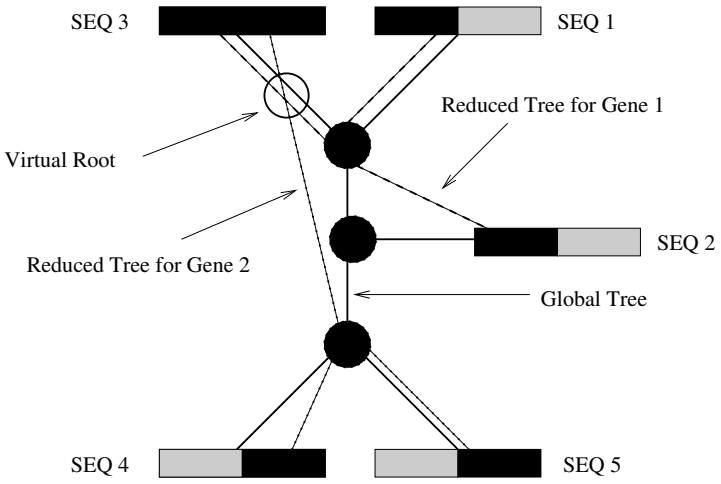


Figure 7.2: Assignment of the gappy sequences to a tree. Per-gene subtrees are connected via a distinct set of pointers for each gene.

The approach presented here is based on three assumptions: i) the data are partitioned on a per-gene basis, ii) a separate set of branch lengths is optimized for every partition, and iii) phylogenomic datasets will remain gappy. While the first assumption provides a computational argument in favor of partitioning phylogenomic alignments on a per-gene basis, the third assumption depends on future developments in wet-lab sequencing technology, but it seems likely that the community will be facing these gappy alignments for at least another five years.

7.3 Static-mesh approach

The usage of static meshes for computing likelihood scores and optimizing branch lengths has already been described [189]. Nonetheless, the concepts introduced in [189] are a prerequisite for developing the update rules for dynamic meshes.

We will initially outline static meshes by example of the multi-gene alignment provided in Figure 7.1. The black regions represent areas of the alignment for which molecular data are available, that is, there is data for 3 taxa in Gene 1 (SEQ 1, SEQ 2, SEQ 3) and for 3 taxa in Gene 2 (SEQ 3, SEQ 4, SEQ 5). The shaded gray regions represent the areas of missing data (undetermined characters). If we assume that the two genes, Gene 1 and Gene 2, have the same length, and 2 out of 5 sequences are missing in each gene, this alignment has a gappyness of 40%.

We can now consider an assignment of these gappy sequences to a fixed tree topology as outlined in Figure 7.2. The comprehensive tree topology that represents the relationships among all 5 taxa for both genes is represented as a thick black line. In order to compute the likelihood for this tree one can, for instance, place the virtual root into the branch of the comprehensive tree that leads to SEQ 3. In order to account for the missing data and omit unnecessary computations, one can use a reduced set of branch lengths and node pointers that only connect those sequences of Gene 1 and Gene 2 (as outlined by the dotted lines in Figure 7.2) for which sequence data are available. This means that we reduce the comprehensive tree topology t to the per-gene topologies $t|G_1$ (read as t restricted to G_1) and $t|G_2$ by successively removing all branches that lead to leaves for which there is no sequence data available in the respective gene. In our example (see also Figure 7.3), the log likelihood of the tree can then simply be computed as $LnL = LnL(t|G_1) + LnL(t|G_2)$. In addition, the memory requirements for storing ancestral probability vectors can be reduced by a factor of 3 in this example, since only one ancestral vector is required for each partition instead of 3 vectors under the standard model.

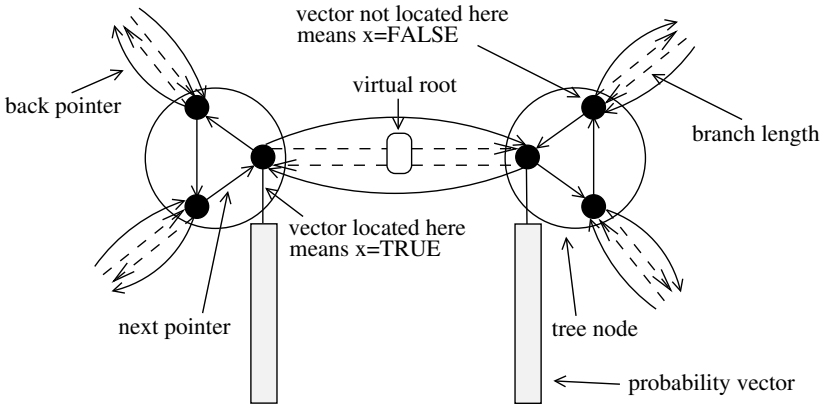


Figure 7.3: Probability vector organization.

Disregarding slight numerical deviations because of round-off errors and a distinct ordering of floating-point operations (floating-point arithmetics are not associative), the above procedure to compute the likelihood on gene-induced subtrees theoretically yields exactly the same likelihood score as the standard method. One should keep in mind that numerical deviations in likelihood scores will increase as more computations are omitted by our approach, because rounding error propagation will become more prevalent.

While the proposed concept is straightforward, the actual implementation is significantly more complicated, especially with respect to an efficient mechanism for computing the topology reduction $t|G_i$ for a gene G_i using the comprehensive topology t. Tree searches, using for instance the SPR technique to optimize tree topologies, need to be appropriately adapted to determine on the fly which ancestral probability vectors for which genes need to be updated. Therefore, a mechanism is required to determine which per-gene subtrees are changed by a SPR move that is applied to the comprehensive tree.

7.3.1 Data structure for per-gene meshes

To describe the implementation of SPR pointer mesh updates in RAxML, we initially need to review the memory and data-structure organization for the single-gene case. The memory space required by standard likelihood implementations is dominated by the length and number of ancestral probability vectors. Thus, the memory requirements are of order $\Theta(n * m)$, where n is the number of taxa and m is the number of distinct site patterns in the alignment. An unrooted phylogenetic tree for an alignment of dimensions $n * m$ has n tips and $n - 2$ inner nodes. Thus, $n - 2$ ancestral vectors of length m are required. Note that the computation of the vectors at the tips of the tree (tip vectors) is significantly less expensive and requires less memory than the computation of inner vectors [28].

In RAxML, only one ancestral probability vector per internal node is allocated. This vector is relocated to one of the three outgoing branches of an internal node `noderec *next` (see data structure below) of the inner node which points toward the current virtual root. This concept is also called a "view" by J. Felsenstein because the ancestral probability vectors always maintain a rooted view of the tree toward the current position of the root. If the ancestral probability vector is already located at the correct outgoing branch (iff. the value of `x == 1`), it does not need to be recomputed. To move ancestral probability vectors among outgoing branches of an inner node, a cyclic list of three data structures of type `node` (one per outgoing branch `struct noderec *back`) is used (see Figure 7.3), which represents a single *ancestral* node of the tree. At all times, two of the entries for `x` in the cyclic list that represents an inner node are set to `x := 0;` whereas the remaining one is set to `x := 1`. The actual probability vector data are then indexed via the node number `number`. Finally, the vector `z[NUM_BRANCHES]` contains the branch lengths for every partition that connects the present node in the cyclic list to the node that is addressed via the respective `back` pointer.

```
typedef struct noderec
{
  double z[NUM_BRANCHES]; /* branch length arrays */
  struct noderec *next;   /* pointer to next structure in cyclic list that
                             represents one internal node */
  struct noderec *back;   /* pointer to neighboring node */
  int number;             /* node number, used to access probability vectors */
  char x;                 /* probability vector located at this node? */
}
node;
```

Using this type of data organization, when the virtual root is placed into a different part of the tree (for instance, to optimize a branch or when the tree topology has been altered), a certain number of ancestral vectors must be recomputed.

The above data structure `node` can be extended as indicated below to accommodate the comprehensive tree topology t as well as the per-gene tree topologies. We extended the data structure by an array of back-pointers `backs` that point to the neighboring nodes for each per-gene tree. Note that the address of the back-pointer of partition 0 for instance, might be located further away, that is, `backs[0] == back` does not necessarily hold. In addition, the array `xs[NUM_BRANCHES]` provides analogous information as `x`, but for each gene separately. By design, if a certain inner node represented by a linked cyclic list of three `node` structures does not form part of a reduced tree $t|G_i$ for gene i, all respective entries are set to `NULL` and `0`: `backs[i] = NULL` and `xs[i] = 0`. If the node does form part of the per-gene tree, all three entries of back-pointers are initialized (`backs[i] != NULL`), and one of the `xs[i]` must be set to 1.

```
typedef  struct noderec
{
  struct noderec *backs[NUM_BRANCHES]; /* back-pointer array */
  char xs[NUM_BRANCHES];               /* probability vector set array */
  double z[NUM_BRANCHES];
  struct noderec *next;
  struct noderec *back;
  int number;
  char x;
}
node;
```

7.3.2 Traversal of a fixed tree

To conduct a tree traversal on a fixed tree for optimizing model parameters and/or branch lengths, we initially place virtual roots for each gene separately. For each gene, we may just place

the virtual root into the first taxon of the respective partition for which data are available. Given a comprehensive tree topology, we recursively set up a per-gene pointer mesh by navigating through the comprehensive tree as described in [189]. The key property of this pointer mesh is that, at any ancestral node, either all outgoing pointers for a specific partition are set to `NULL` or all outgoing pointers point to another node in the tree that contains data for the specific gene. In other words, a node of the comprehensive tree either forms part of the per-gene subtree or not.

To compute the likelihood and optimize branch lengths as well as model parameters for the fixed comprehensive tree, we simply need to execute Felsenstein's pruning algorithm individually on every gene by using the respective pointer meshes and branch lengths induced by `*backs[NUM_BRANCHES]` and `z[NUM_BRANCHES]`. The dotted lines in Figure 7.2 indicate the reduced tree data structures given by the `backs[]` arrays (the traversal path for a gene tree), while the straight line represents the overall tree topology as provided by the `back` pointers. To achieve the desired memory reduction for ancestral probability vectors (in contrast to the initial paper [189] where this was not implemented), for each gene we only allocate as much memory as required for the number of ancestral nodes contained in the gene tree. Since we address ancestral vectors via the node `number`, this means that in the course of computations per-gene trees are always represented by the same nodes in memory, that is, a node that once formed part of a gene tree will always from part of that gene tree.

The branch length optimization procedure works analogously, with the only difference that `xs[NUM_BRANCHES]` are also updated, since optimizing the branch lengths of a tree induces continuously re-rooting the tree at the branch to be optimized. Thus, one can easily use the above data structure to successively and individually optimize the branches in every gene tree. For a more detailed description refer to [189]. Finally, it is important to note that the speedups reported here are smaller than those reported previously [189], because of a bug in the branch length optimization convergence criterion under the standard model. This bug has been fixed in the latest release of RAxML (as of version 7.2.5).

7.4 Dynamic-mesh approach

Given the prolegomena, we can now introduce a set of rules for per-gene pointer mesh updates that are required to conduct SPR moves on the above data structure. For this, we need to consider the pruning (removing a subtree at a specific branch of the *comprehensive* tree) and regrafting (inserting the pruned subtree into branches of the tree) steps separately.

7.4.1 Subtree pruning

If we prune a subtree from a branch of the comprehensive tree as shown in Figure 7.4, we have to consider the following cases:

Case 1: If the subtree to be pruned does not contain any data for gene i, the induced

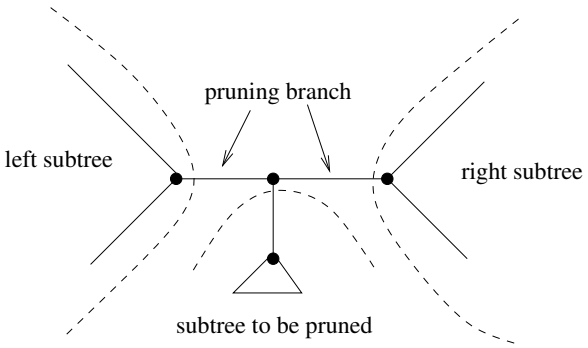


Figure 7.4: Pruning of a subtree.

gene tree will be invariant with respect to any SPR moves of that subtree, and hence the log likelihood $LnL(t|G_i)$ will be invariant as well. Therefore, we just need to set a respective flag for gene i and add the current likelihood for partition i to the likelihood of the other genes that may change because of the SPR move. We can check if the subtree does not contain data for gene i by a simple recursive descent for partition i using the corresponding pointer mesh of that subtree.

Case 2: If the node to which the subtree is attached in the comprehensive tree forms part of a gene i, that is, if all `backs[i]` are not set to `NULL`, we store the values of all three outgoing `backs[i]` pointers and remove the node from the per-gene pointer mesh.

Case 3: If the node to which the subtree is attached in the comprehensive tree does not form part of the gene tree i, we initially determine whether data for gene i is contained in the left or right subtree rooted at the branch from which the subtree is pruned. If neither the left nor the right subtree contains data for gene i, this means that the gene tree for i is entirely contained in the subtree to be pruned. Therefore, we once again set a respective flag for gene i that there is no work to be done and store the current likelihood score for this partition. Otherwise, either the left or the right or both subtrees will contain data for partition i. In the case that both subtrees of the pruning branch as well as the subtree to be pruned contain data, we connect the nodes in the left and right subtrees and prune the subtree. If only one of the two subtrees defined by the pruning branch contains data, this means that the subtree to be pruned is directly connected to a node in one of these two subtrees.

In all cases that there is work to do, we always store the nodes that define the left and the right end of the branch (the pruning branch) from which the subtree is to be removed. Finally, after pruning the subtree, the branch length of the pruning branch from which the subtree was removed is optimized via a Newton-Raphson procedure. Once we have pruned the subtree and stored the required data, we can start reinserting the pruned subtree into the remainder of the comprehensive tree.

7.4.2 Subtree regrafting

The SPR moves are initially conducted on the comprehensive tree data structure. For each insertion branch in the comprehensive tree into which the candidate subtree shall be inserted, we once again conduct a case analysis to determine if the specific SPR move on the comprehensive tree also induces a change in the gene trees. For the cases described in Section 7.4.1 where the pruned subtree either contains all taxa of a gene or not a single taxon of a gene i, there is no work to do because the SPR move will not induce any changes to $LnL(t|G_i)$. Thus, in this case we simply add the stored likelihood of the gene to the overall likelihood we intend to compute.

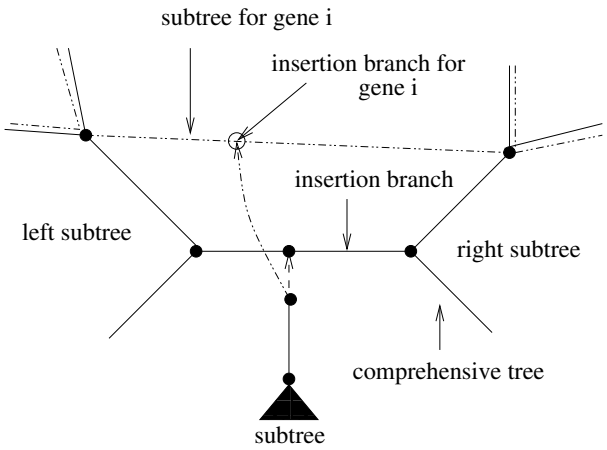


Figure 7.5: Subtree insertion into a branch. Both the left and right subtrees contain data for a specific gene i.

In all other cases, the SPR move on the comprehensive tree *may* induce changes to the gene tree. If we consider the insertion branch, that is, the branch of the comprehensive tree into which the subtree shall be inserted, we once again need to determine recursively whether the left and/or the right subtrees of the insertion branch contain molecular data for a partition i (see Figure 7.5).

Case 1: If some nodes in the left *and* the right subtrees of the insertion branch contain data for partition i as outlined in Figure 7.5, this means that the two subtrees must be connected by a branch of the partition, the partition insertion branch, into which the subtree shall be inserted. We obtain the per-gene insertion branch by recovering the two nodes that determine this branch via a recursive descent into the left and right subtrees of the insertion branch in the comprehensive tree.

However, given that each gene tree contains at most as many ancestral nodes as the comprehensive tree, it may happen that the insertion branch for the subtree of partition i is identical (and hence the insertion likelihood is identical) for distinct insertion branches in the comprehensive tree (see Figure 7.6). Thus, the likelihood that is induced by repeatedly inserting the

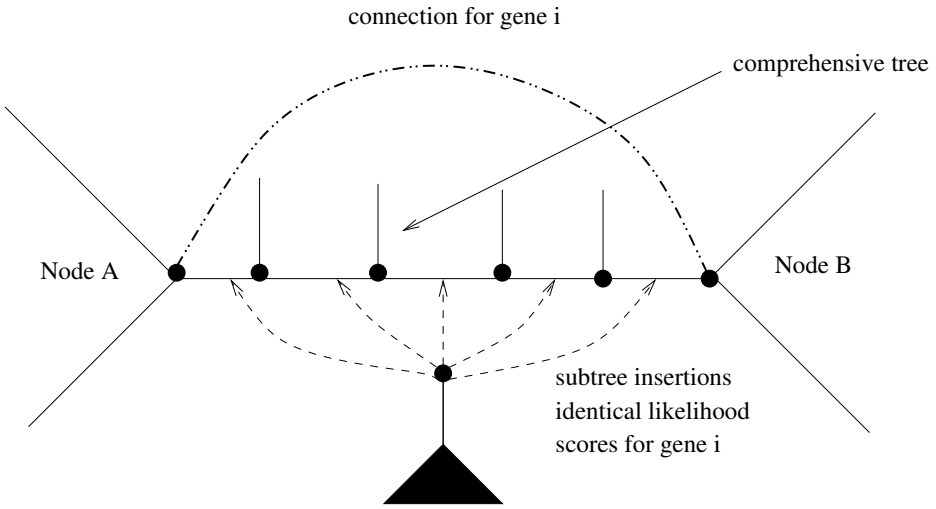


Figure 7.6: Outline of the optimization potential for subtree insertions into consecutive branches that have identical left and right subtrees for a specific gene i.

gene subtree i between the two nodes `A` and `B` that form a branch of the gene tree is identical. In other words, for the five subtree insertions between nodes `A` and `B`, the tree topology induced for the gene tree i is invariant. Thus, the likelihood score for insertions of subtree i between nodes `A` and `B` only needs to be computed once. To achieve computational savings, we use a simple linked list to store and look up the gene tree insertion branch nodes/likelihood score triplets. For instance, at the first insertion of the subtree between nodes `A` and `B`, for gene i we will look up if the node pair `A, B` is stored in the respective list for partition i. If this is not the case, we will compute the likelihood score and store it in the list. For all successive insertions of the per-gene subtree between nodes `A` and `B`, the lookup for partition i will be successful, and we can hence simply reuse the per-gene likelihood score instead of recomputing it.

Case 2: If only a node to the left *or* the right of the comprehensive insertion branch contains data for a partition i, this means that a subtree for gene i is located in one of the two subtrees, but not both. In this case, we recursively descend into the subtree that contains the data until we find the first node that belongs to the gene tree i. If the node is not a tip, it must be connected to another node in the same subtree of the comprehensive tree as shown in Figure 7.7. As before, we determine whether the candidate subtree has already been inserted into the respective branch of the gene tree by conducting a lookup in the aforementioned linked list. For example, in Figure 7.7, the insertion likelihood for gene i is identical when the subtree is inserted into branches `X`, `Y`, or `Z` of the comprehensive tree. If this is not the case, we insert the candidate subtree, compute its likelihood, and add it to the likelihood scores of the remaining genes.

Finally, if either the left or the right subtree of the comprehensive insertion branch contains

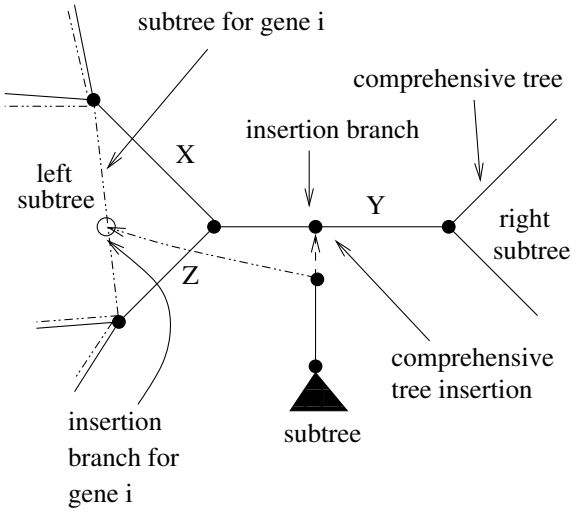


Figure 7.7: Insertion of a subtree into a branch at which only one of the two subtrees contains data for gene i.

only a single taxon for gene i, this taxon represents the position from which the gene subtree was pruned. This case was already detected at the pruning stage, and an appropriate flag was set to indicate that the likelihood for partition i does not need to be recomputed for any subtree insertion.

7.5 Experimental evaluation

7.5.1 Implementation

The pointer mesh update rules were implemented in the sequential SSE3-vectorized version [33] of RAxML (freely available in the latest standard-RAxML release at `https://github.com/stamatak/standard-RAxML`; mesh-based methods are implemented in file `mesh.c`).

To facilitate verification and comparison of the results, we implemented a simplified version of the lazy SPR move technique in RAxML [182]. The lazy SPR move technique of RAxML works as follows. Initially, the subtree to be rearranged (the candidate subtree) is pruned from the comprehensive tree topology. Then, only the branch from which the subtree was pruned is reoptimized via a Newton-Raphson procedure, as opposed to reoptimizing all branch lengths in the remaining tree. After this step, we can start inserting the candidate subtree into the branches of the remaining tree and compute the likelihood score for each insertion. When lazy SPR moves are used, instead of reoptimizing all branches of the resulting tree, we only reoptimize the three branch lengths that are adjacent to the insertion position of the candidate subtree. Thereby, we only obtain an approximate likelihood score instead of a maximum likelihood score for each subtree rearrangement, that is, we conduct a lazy evaluation of SPR moves. The fast and slow

lazy SPR moves implemented in RAxML differ in the way the three branch lengths adjacent to the subtree insertion position are optimized. Fast lazy SPR moves use an empirical best guess for the three branches, while slow lazy SPR moves deploy the Newton-Raphson procedure. Note that other widely used ML-based inference programs, like PHYML v3.0 and GARLI, also use variants of lazy SPR moves. The lazy SPR move technique is outlined in Figure 7.8.

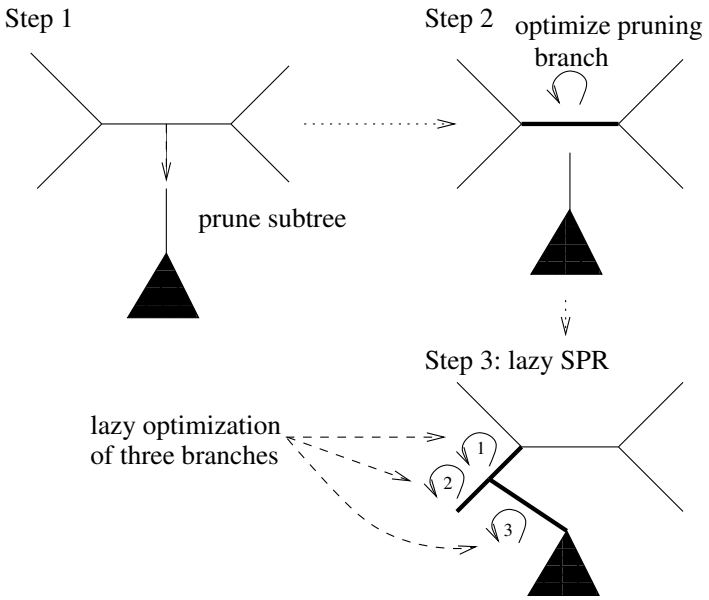


Figure 7.8: Lazy SPR move technique of the standard RAxML algorithm.

The simplified version of the lazy SPR move technique implemented here reads in a given starting tree, optimizes model parameters as well as branch lengths, and then applies only one cycle of lazy SPR moves to the tree with a rearrangement radius that is fixed to 10. One cycle of SPR moves means that every subtree of the comprehensive tree will be pruned and reinserted into all neighboring branches of the pruning branch up to a distance of 10 nodes away from the original pruning position.

In addition, unlike the standard RAxML search mechanism, the algorithm will not immediately keep SPR-generated topologies that yield an improvement, since it only searches for the best lazy SPR move on the tree that was provided as input. The algorithm stores the best move on the comprehensive tree and will then write the comprehensive tree generated by the best SPR move to a file. For verification purposes, this output tree can then be used to independently compute likelihood scores on the tree topologies obtained by the standard method and the fast mesh-based method we propose here.

The program will also print out the execution time required for model optimization, the execution time of one SPR cycle, and the overall execution time. The lazy SPR searches can be executed using fast lazy insertions and slow/thorough lazy insertions [182]. The respective

command lines for the mesh-based approach are:

```
./raxmlHPC-SSE3 -f i -C -M -s alignment.phy -t startingTree -q partitions
                -m GTRGAMMA -n MESH_FAST
./raxmlHPC-SSE3 -f I -C -M -s alignment.phy -t startingTree -q partitions
                -m GTRGAMMA -n MESH_THOROUGH,
```

while for the standard approach are:

```
./raxmlHPC-SSE3 -f i -M -s alignment.phy -t startingTree -q partitions
                -m GTRGAMMA -n NO_MESH_FAST
./raxmlHPC-SSE3 -f I -M -s alignment.phy -t startingTree -q partitions
                -m GTRGAMMA -n NO_MESH_THOROUGH.
```

The inferences were conducted under the GTR [111] nucleotide substitution model and the WAG [204] amino acid substitution model for protein data. In all cases we used the standard Γ model of rate heterogeneity [210]. The computational experiments were executed on a single core of an unloaded SUN x4600 multi-core machine with 32 cores and 64 GBs of main memory. The program was compiled with `gcc v4.3.2` and the standard makefile for the sequential SSE3 code that is distributed with the source code.

7.5.2 Datasets

We used 6 real-world DNA and protein datasets containing 59 up to 37,831 taxa and 6 up to 1,487 genes. The gappyness because of missing gene data ranged between 27% and 90%. Table 7.3 indicates the gappyness of the alignments used (not counting real alignment gaps). For ease of reference, we denote all datasets by dY_X, where Y indicates the number of taxa and X indicates the number of genes. Dataset d94_1487 is a protein alignment [88]. All other datasets are DNA alignments. All datasets (except for the unpublished dataset d37831_6), partition files, and starting trees are available for download at `http://www.exelixis-lab.org/pointerMeshData.tar.bz2`.

7.5.3 Results

The recursive lookups to search for gene nodes in the pruning branch and insertion branch subtrees are implemented by recursive descents into the subtrees. While this is algorithmically not very elegant, the efficiency of this procedure is not critical, since a profiling run using `gprof` on datasets d59_8 and d404_11 revealed that the recursive search procedures account for less than 1% of total execution time. On dataset d37831_6 the contribution may be higher, but a profiling run could not be conducted because of excessive runtimes and the significant slowdown associated with profiling.

In Table 7.1, we indicate the execution time speedups between the standard implementation and the mesh-based approach for model parameter optimization (denoted as Model Optimization) as well as the fast lazy (denoted as Fast SPR) and the slow lazy SPR searches (denoted as Slow SPR). Overall, speedups for the fast lazy SPRs tend to be higher than for the thorough lazy SPRs, particularly on protein data.

Dataset	Model Optimization	Fast SPR	Slow SPR
d59_8	1.30	2.04	1.59
d94_1487	5.56	16.69	4.41
d126_34	1.34	1.79	1.80
d404_11	3.05	4.91	3.51
d2177_68	11.24	16.08	10.26
d37831_6	3.86	5.36	3.99

Table 7.1: Speedups of mesh-based approach versus standard approach.

In Table 7.2, we provide the overall execution times in seconds (including file I/O, model optimization, and SPR searches) for the mesh-based and the standard (denoted as NoMesh) likelihood function implementations using the fast and the more thorough lazy SPR moves.

Dataset	Fast Mesh	Fast NoMesh	Slow Mesh	Slow NoMesh
d59_8	21	32	74	114
d94_1487	7,493	77,960	92,573	408,794
d128_34	1,106	1,592	2,741	4,523
d404_11	159	597	597	2,066
d2177_68	7,395	87,320	15,455	164,168
d37831_6	31,597	130,776	94,497	367,139

Table 7.2: Total execution times in seconds of mesh-based approach versus standard approach.

In Table 7.3, we provide the gappyness of each dataset, that is, the proportion of entirely missing data per gene over the entire alignment, and the memory footprint for inferences under GTR+Γ and WAG+Γ for the mesh-based and standard approaches. The memory savings are roughly proportional to the degree of gappyness.

Finally, in Table 7.4, we depict the likelihood scores of the trees computed independently by optimizing the likelihood score on the resulting SPR-modified trees obtained by the mesh-based and standard methods. The scores on the trees were optimized using the mesh-based approach to save time, but for the smaller datasets we also conducted a tree evaluation using the standard approach. As already mentioned, likelihood scores may be slightly different (because of numerical deviations) when model parameters are optimized using the standard approach. It is interesting to observe that for thorough lazy SPR moves, the mesh-based approach yields

Dataset	Gappyness	Memory NoMesh	Memory Mesh
d59_8	27.70%	25 MB	19 MB
d94_1487	81.31%	14.0 GB	2.8 GB
d128_34	28.30%	317 MB	234 MB
d404_11	69.15%	378 MB	125 MB
d2177_68	89.53%	9.0 GB	1.1 GB
d37831_6	75.41%	44.0 GB	14.0 GB

Table 7.3: Gene-sampling-induced gappyness and memory consumption of mesh-based approach versus standard approach.

Dataset	Fast Mesh	Fast NoMesh	Slow Mesh	Slow NoMesh
d59_8	-50439.82	-50439.82	-50434.80	-50434.80
d94_1487	-5996718.37	-5996718.37	-5996650.63	-5996707.99
d128_34	-779459.01	-779459.01	-779446.71	-779446.71
d404_11	-151064.76	-151064.76	-151064.76	-151064.76
d2177_68	-2166752.48	-2166752.48	-2166237.10	-2166433.84
d37831_6	-5418619.45	-5418619.45	-5418648.55	-5418648.55

Table 7.4: Log likelihoods of final trees generated by mesh-based and standard approaches using fast and slow SPR cycles.

slightly better likelihood scores on datasets d2177_68 and d94_1487, which can be attributed to reduced numerical error propagation. When the standard approach is used, a significantly larger number of computations is conducted, which may introduce rounding errors.

7.6 Summary

This chapter presented a novel technique and an open-source implementation for conducting tree searches on phylogenomic alignments with missing data. A set of rules were introduced to dynamically update per-gene subtrees while applying SPR tree rearrangement operations on a comprehensive tree.

By deploying this rule set, we can significantly reduce the number of floating-point operations required to compute the phylogenetic likelihood function and thereby accelerate likelihood-based tree searches by up to one order of magnitude. More importantly, our method also allows for a memory footprint reduction that is proportional to the gappyness (proportion of missing data) of the alignment. Large and computationally challenging phylogenomic analyses under likelihood, which would otherwise require supercomputers, can now be conducted on a desktop computer. The methods presented here can be applied to *all* likelihood-based (ML and Bayesian) programs to accelerate computations on typical phylogenomic alignments. We have demonstrated that

we can achieve memory and time savings of one order of magnitude by deploying our rule set on datasets with a gappyness of approximately 90%.

Chapter 8

Selective Sweep Detection in Whole-Genome Datasets

This chapter presents a scalable implementation for computing the ω statistic that can be used to efficiently detect selective sweeps in very large population genetic datasets based on linkage-disequilibrium patterns. Furthermore, three parallelization schemes (fine-grain, coarse-grain, and multi-grain) are described and evaluated.

8.1 Introduction

The field of population genetics studies the genetic composition of populations as well as changes in this genetic composition that are, for instance, driven by natural selection or genetic drift [48]. Positive or directional selection refers to a mode of natural selection that favors a certain phenotypic direction (e.g., increased resistance to antibiotics). If an allele contributes to the phenotypic shift that is favored by selection, its frequency will increase in the population and eventually it will fix (all individuals in the population will have this allele). When a strongly beneficial allele occurs and spreads in a population, it is inevitable that the frequency of linked neutral mutations will increase. Maynard Smith and Haigh [127] termed this process 'genetic hitch-hiking'. They showed that in very large populations, hitch-hiking can drastically reduce neutral genetic variation near the site of selection, thus causing a selective sweep. Apart from the reduction of neutral variation, genetic hitch-hiking also modifies the frequency of neutral mutations [39] as well as their linkage-disequilibrium values (see Section 8.2). A major goal of modern population genetics is to analyze neutral genetic variation and identify patterns (neutral polymorphic patterns) that correspond to the aforementioned predictions of the hitch-hiking effect. These patterns are often referred to as signatures of a selective sweep.

Statistical tests that rely on selective sweep theory [127] are widely used to identify targets of recent and strong positive selection. This is achieved by analyzing single nucleotide polymorphisms (SNPs) in intra-species MSAs. A SNP refers to variations observed in a single

column of an intra-species MSA. These variations denote that, at specific genomic positions, nucleotides differ among the members of the species. Several studies in the last decade have focused on detecting positive selection in genomes of natural populations of individuals. Kim and Stephan [106] developed a composite maximum likelihood (ML) framework to detect selective sweeps using the empirical mutation frequencies of polymorphic sites (site frequency spectrum). This approach was later adapted by Nielsen *et al.* [139] for populations that have experienced past demographic changes (e.g., changes in population size). Kim and Nielsen [105] proposed the ω statistic to accurately localize selective sweeps and developed a ML framework that uses linkage-disequilibrium (LD) information. The ω statistic [105] has been implemented by Jensen *et al.* [95] and Pavlidis *et al.* [154].

In this chapter, we introduce OmegaPlus (Section 8.3), a high-performance, open-source (`http://www.exelixis-lab.org/OmegaPlus_v1_Linux.php`) implementation of the ω statistic. The kernel function of OmegaPlus, which dominates execution times, consists of a dynamic programming algorithm for calculating sums of LD values in sub-genomic regions (fractions of the genome). To the best of our knowledge, OmegaPlus represents the first scalable implementation of the ω statistic that *can* be applied to whole-genome data. Previous implementations were not suitable for whole-genome analyses: the implementation by Jensnen *et al.* [95] is only suitable for analyses of sub-genomic regions, while the implementation by Pavlidis *et al.* [154] (Omega software tool) has excessive memory requirements even for moderate datasets. OmegaPlus has lower memory requirements than competing codes and can analyze whole-genome MSAs on off-the-shelf multi-core desktop systems.

Thereafter, we developed two parallel versions: i) OmegaPlus-F (Section 8.4.1), which parallelizes the dynamic programming matrix computations at a fine-grain level and ii) OmegaPlus-C (Section 8.4.2), which divides the alignment into sub-regions and assigns these individual and independent tasks to threads (coarse-grain parallelism). These relatively straightforward parallelization schemes yield acceptable, yet sub-optimal speedups because both approaches exhibit some drawbacks. Exploiting fine-grain parallelism in dynamic programming matrices, especially when these are relatively small, can exhibit unfavorable computation-to-synchronization ratios. This is the case in OmegaPlus-F, despite the fact that an efficient thread synchronization strategy based on busy waiting [34] is used. Thus, because of the relatively small amount of operations required per dynamic programming matrix cell, the fine-grain parallel version only scales well if the input MSA comprises hundreds to thousands of sequences. The coarse-grain OmegaPlus-C version also faces a parallel scalability problem: the variability of SNP density along the MSA can cause substantial load imbalance between threads. Despite the fact that all sub-genomic regions are of the same size, tasks that encompass regions with high SNP density require significantly longer execution times. Note that the time required to compute the ω statistic in a sub-genomic region (a task) increases quadratically with the number of SNPs in that region. Hence, the parallel performance of OmegaPlus-C is limited by the slowest thread/task, that is, the thread assigned to the sub-genomic region with the highest SNP density.

To alleviate this load imbalance issue, Section 8.4.3 introduces a multi-grain parallelization strategy that combines the coarse- and fine-grain approaches. The underlying idea is to use fast threads that have completed their task to accelerate the ω statistic computations of slower threads in a fine-grain way. In other words, when a thread completes its coarse-grain task early, it is assigned to help the thread that is expected to finish last at that particular point in time. Multi-grain parallelization approaches have also been used for evolutionary placement of short reads [187] and for computing the phylogenetic likelihood function on the Cell Broadband Engine [36]. In the following section, we describe the LD pattern of selective sweeps.

8.2 Linkage-disequilibrium (LD) pattern of selective sweeps

The LD is used to capture the non-random association of states at different MSA positions. Assume two sites i and j with two states each: i_1, i_2 for site i and j_1, j_2 for site j. LD quantifies whether a combination of states, e.g., $i_1 j_1$, occurs in the same sequence more often or less often than expected by chance. In Figure 8.1, states i_1 and j_1 are depicted as circles, while states i_2 and j_2 are depicted as squares. LD quantifies the tendency of circles at site i (i_1) to coexist with circles at site j (j_1) on the same line, i.e., same sequence/individual. The selective sweep theory [127] predicts a pattern of excessive LD in each of the two alignment regions that flank a recently fixed beneficial mutation. This genomic pattern can be detected by using the ω statistic.

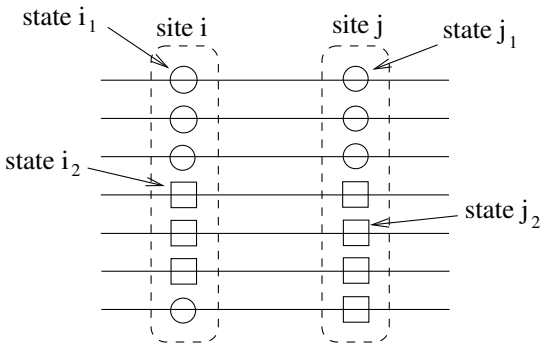


Figure 8.1: Example of two polymorphic sites with two different states each. The states are depicted as circles and squares.

Several measures for computing the LD exist, such as D [116] and D' [115]. Today, the most commonly used measure to capture LD is the squared correlation coefficient (r^2). If the states in each site are encoded by ones and zeros, then the LD can be easily measured by representing each site as a bit vector and computing the squared correlation coefficient r_{ij}^2 between two sites i and j as follows:

$$r_{ij}^2 = \frac{(x_{11} - p_1 q_1)^2}{p_1 q_1 (1 - p_1)(1 - q_1)},$$

where x_{11} is the frequency of combinations with $i_1 = 1$ and $j_1 = 1$, p_1 is the frequency of ones in site i, and q_1 is the frequency of ones in site j.

Figure 8.2 shows the generation of SNP patterns that can be used to locate a selective sweep. The figure comprises 6 snapshots that depict a population of chromosomes at different points in time. The horizontal lines correspond to homologous chromosomes from different individuals. Snapshot 1 is the oldest while snapshot 6 is the most recent. Thus, in snapshot 1, neutral mutations were segregating in a population. At some point in time (snapshot 2), a beneficial mutation appears (black circle). Since this mutation is beneficial, the frequency of the chromosome that carries it will increase in the population (snapshot 3). However, recombination between the beneficial chromosome and the neutral chromosomes may also occur. In snapshot 4, recombination occurs on the left-hand side of the beneficial mutation, while in snapshot 5, recombination occurs on the right-hand side. Finally, in snapshot 6, we highlight the regions where LD between pairs of SNPs is high on the left- and right-hand side of the beneficial mutation. The LD between pairs of SNPs that are located on different sides of the beneficial mutation is low.

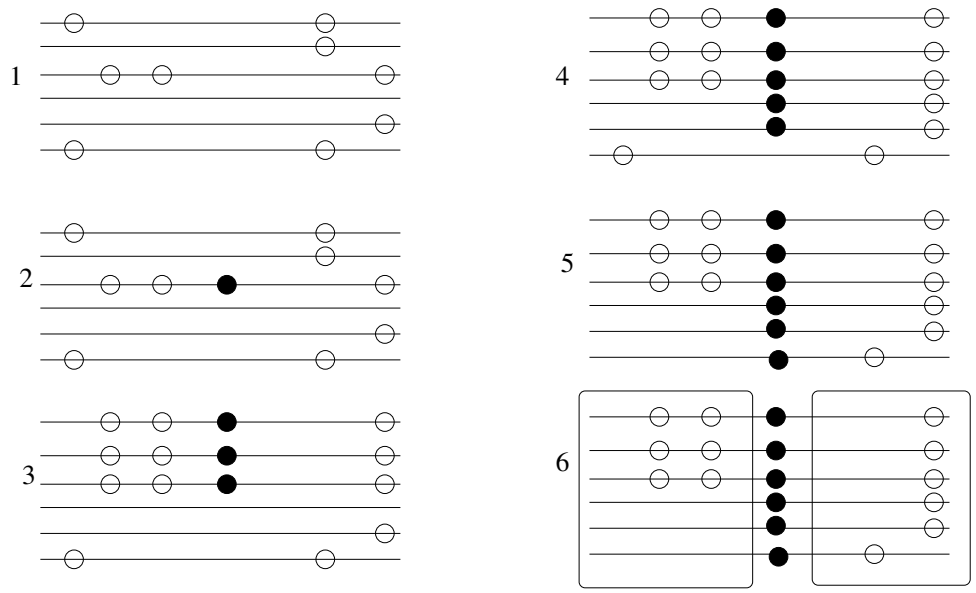


Figure 8.2: LD patterns generated by a selective sweep. 1. Neutral mutations (light circles) are present in the population. 2. A beneficial mutation (black circle) appears in the population. 3. The frequency of the chromosome that carries the beneficial mutation increases. 4. Due to recombination (between chromosomes 3 and 4), neutral mutations that were previously on a neutral chromosome are now located on a beneficial chromosome. 5. Recombination occurs between chromosomes 3 and 6, which adds other neutral mutations to the beneficial chromosome. 6. Square regions on the left and right side of the beneficial mutation denote the regions where the values of LD between SNP pairs are high. Linkage disequilibrium between pairs of SNPs that are located on different sides of the beneficial mutation is low.

Assume a genomic window with S SNPs that is split into a left sub-region L and a right sub-region R with l and $S-l$ SNPs, respectively. The ω statistic is then computed as follows:

$$\omega = \frac{(\binom{l}{2} + \binom{S-l}{2})^{-1}(\sum_{i,j \in L} r_{ij}^2 + \sum_{i,j \in R} r_{ij}^2)}{(l(S-l))^{-1} \sum_{i \in L, j \in R} r_{ij}^2}. \tag{8.1}$$

The ω statistic quantifies to which extent average LD is increased on both sides of the selective sweep (see numerator of Equation 8.1) but *not across* the site of the beneficial mutation (see denominator of Equation 8.1). The area between the left and right sub-regions is considered as the center of the selective sweep.

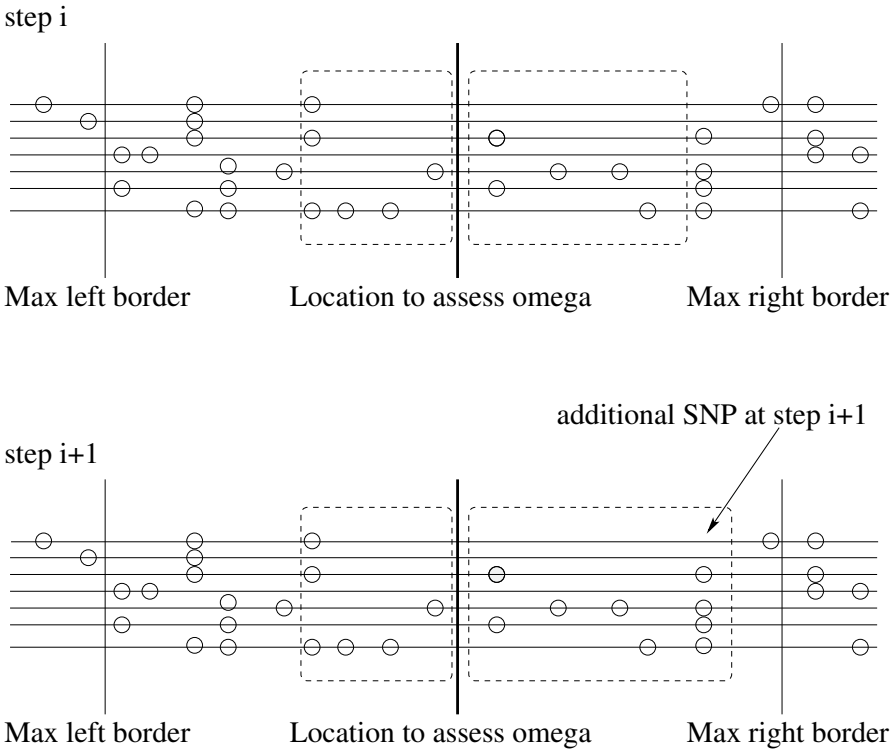


Figure 8.3: The process of detecting the sub-regions that maximize the ω statistic for a given location/selective sweep. The goal is to evaluate the ω statistic at the alignment position denoted by the thick vertical line. The thin vertical lines to the left and right side of the thick line indicate the sub-region borders as defined by R_{MAX}. At step i, only the SNPs enclosed within the dashed-line areas contribute to the calculation of ω statistic. At step $i+1$, one more SNP that belongs to the right sub-region contributes to the ω statistic calculation. Calculations are repeated for all possible sub-regions within the left and right borders (vertical thin lines). The maximum ω value and the associated sub-region sizes are then reported.

In sub-genomic regions, that is, candidate regions of limited length (some thousand bases/sites long), the ω statistic can be computed at each interval between two SNPs. S refers to the total number of SNPs, and the goal is to detect that l which maximizes the ω statistic.

When scanning whole-genome datasets, the analysis becomes more complicated. Evaluating the ω statistic at each interval between two SNPs can become computationally prohibitive since hundreds of thousands of SNPs may occur in an entire chromosome. Furthermore, S can not be defined as the overall amount of SNPs along the whole chromosome. This would also be biologically meaningless because a selective sweep usually only affects the polymorphic patterns in the neighborhood of a beneficial (advantageous) mutation.

To process whole-genome datasets, we assume a grid of equidistant locations L_i, $1 < i < N$; N is defined by the user. The ω statistic is computed at each grid position L_i. We also assume a user-defined sub-genomic region size R_{MAX} that extends to either side of a beneficial mutation. Such a sub-genomic region represents the genomic area (neighborhood) in which polymorphic patterns may have been affected by the selective sweep. Finally, we evaluate the ω statistic for all possible sub-regions that are enclosed in R_{MAX} and report the maximum ω value as well as the sub-region size that maximizes ω. Figure 8.3 illustrates two consecutive steps of the ω statistic calculation at a specific location. The user-defined R_{MAX} value determines the left and right borders (vertical thin lines). The ω statistic is computed for all possible sub-regions that lie within the left and right borders, and the maximum ω value is reported. The process is repeated for all locations for which the ω statistic needs to be computed.

8.3 OmegaPlus software

OmegaPlus is open-source, implemented in C, and can be compiled on Windows and Linux platforms.

8.3.1 Computational workflow

OmegaPlus calculates the ω statistic at N distinct positions in the alignment. For each position p, $1 \leq p \leq N$, let k_L and k_R be the numbers of SNPs to the left and to the right of p, respectively, which are contained within a user-defined genomic region G. Thus, a number of k_L and k_R regions are located to the left and right side of p, each containing a different number of SNPs. The ω statistic is computed for all $k_L \times k_R$ pairs of sub-regions, and the pair for which ω statistic has the highest value is reported.

An overview of the computational workflow of OmegaPlus on whole-genome datasets is provided in Figure 8.4. The figure shows the basic steps for the calculation of an ω statistic value at a given position. The same procedure is repeated to compute ω statistic values at all positions p that have been specified by a user parameter. Initially, we check whether sub-genomic regions, as defined by R_{MAX}, overlap. If they overlap, the data are re-indexed to avoid recalculation of previously computed values and thereby save computations. Thereafter, all required new LD values and all sums of LDs in the region are computed. Finally, all possible ω statistic values for the specific region are computed.

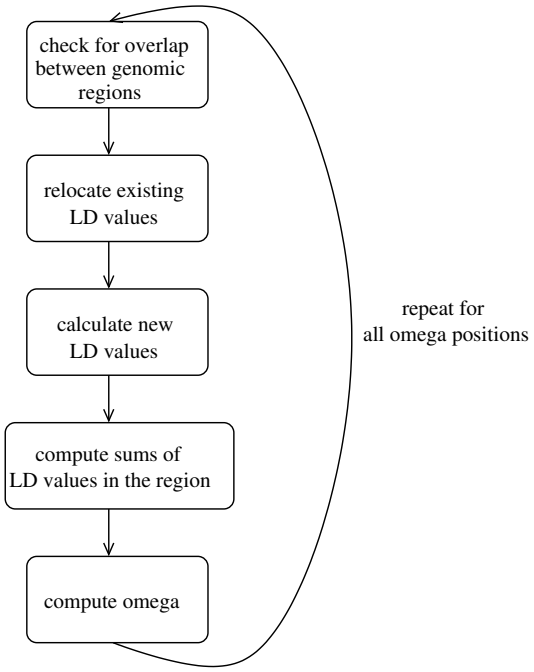


Figure 8.4: The basic algorithmic steps of OmegaPlus for the computation of the ω statistic at a number of alignment positions requested by the user.

8.3.2 Input/Output

OmegaPlus is a command-line tool that only requires five input arguments for a typical run: an alignment file (`-input`), the number p of positions at which the ω statistic will be calculated (`-grid`), a minimum (`-minwin`) and maximum (`-maxwin`) size for the L and R sub-regions, and a name for the specific run (`-name`). Binary data (derived/ancestral states, see Section 8.4.1) in ms [91] or MaCS [44] format and DNA data in FASTA format can be analyzed. For ms and MaCS files, the length of the alignment (`-length`) also needs to be provided by the user. For example, the tool can be launched as follows:

```
./OmegaPlus -name test -input alignment.fa -minwin 100 -maxwin 1000 -grid 5000,
```

to process the `alignment.fa` file, calculate the ω statistic at 5,000 alignment positions p, and calculate all $k_L - k_R$ sub-region pairs of sizes between 100 and 1,000 alignment sites.

The program generates three text files: an information file, a warning file, and a report file. The information file contains details about the execution of the program, such as the command line used and the total execution time. The warning file reports conflicting SNP positions in the alignment, that is, SNPs which refer to the same alignment positions (this may only occur when binary data are analyzed). Finally, the report file provides the highest ω statistic value for each position p. A detailed description of the input and output options is provided in the manual [6].

8.3.3 Memory requirements

One primary design goal was to reduce memory requirements. In general, two factors determine the memory footprint of Omega [154] and OmegaPlus: the size of the input alignment *and* a two-dimensional matrix that is used to store the linkage-disequilibrium values of a genomic region.

The input alignment size increases in proportion to the number of sequences and number of sites. A binary alignment of 1,000 sequences and 1,000,000 SNPs for example, requires 960 MBs of RAM. OmegaPlus uses arrays of unsigned integers to store the input alignment in a compressed form. This approach not only reduces the amount of memory required to store the alignment but also facilitates the calculation of the linkage-disequilibrium values. The elements at an alignment site (0, 1 for binary data and A, C, G, T for DNA data) are stored in unsigned integers as group of 32 elements in each unsigned integer. For the 1,000-sequence example, an alignment site consists of 1,000 elements, and therefore 32 unsigned integers are required to store one site. This simple bit-wise compression reduces the memory required to store the alignment from 960 MBs to only 121 MBs. The memory demands may slightly increase if the alignment consists of gaps (-) and/or ambiguous characters (M, N, etc.). In this case, more memory is required to store the positions of the gaps and ambiguous characters in the alignment.

In addition to the input alignment, the LD values in a genomic region of the alignment must be temporarily stored since they are used again later-on for the computation of the ω statistic in this genomic region. The size of genomic regions varies because it depends on user parameters (alignment length and window size). Furthermore, the amount of memory required to store the linkage-disequilibrium matrix also depends on the number of SNPs in a given region. For instance, if a genomic region contains 20,000 SNPs, 763 MBs of memory are required to store the LD values. Note that the LD values are floating-point numbers.

To further reduce the memory demands, OmegaPlus analyzes only the SNPs flanking a position at which the ω statistic will be computed. This technique resolves the memory-related problems of Omega [154]. In Omega, the user is required to define the size of the genomic regions into which the alignment should be split. This user-initiated choice leads to the following problem: if the genomic regions are not large enough, ω values at positions near the borders of the region will not be calculated correctly. On the other hand, if the regions are much larger than required, the program execution time will increase dramatically. OmegaPlus solves this problem by initially scanning the alignment to detect the maximum number of SNPs that will be used to compute the value of each ω along the alignment. This pre-processing of the alignment is done on a per-L_i basis, and the memory footprint of the program depends on the area with the highest SNP density around an ω position. If the most SNP-dense genomic region (around an ω position) contains 10,000 SNPs for instance, OmegaPlus will allocate a memory block of $(10,000^2)/2$ floating-point values (191 MBs) to process the specific area. OmegaPlus will subsequently reuse the same allocated memory block to process all other (smaller) regions of the alignment. Due to the quadratic increase in memory requirements with the number of SNPs

in a genomic region, the OmegaPlus source code contains a hard-coded value for the maximum allowed number of SNPs in a region. To ensure scalability of the code, this upper bound value is set to 10,000 SNPs. When, as per user input, OmegaPlus needs to process a genomic region that contains more than 10,000 SNPs, the window size parameter (set by the user with `-maxwin`) is reduced to the maximum value complying with the 10,000-SNP restriction. Alternatively, one can increase this hard-coded limit in the source code (constant `MAXSIZE` in OmegaPlus.h).

8.3.4 Sum of LD values in sub-genomic regions

To accelerate the calculation of ω statistic values, a two-dimensional matrix M, $M_{l,m} = \sum_{l \leq i < j \leq m} r_{ij}^2$, is calculated for each region G. Thereafter, all $\sum_{i \in L, j \in R} r_{ij}^2$, $\sum_{i,j \in L} r_{ij}^2$, and $\sum_{i,j \in R} r_{ij}^2$ values required by Equation 8.1 can be retrieved from M. This technique avoids redundant calculations and keeps the memory requirements proportional to k^2, where k is the total number of SNPs in G. For this purpose, OmegaPlus introduces a dynamic programming algorithm that calculates M with time complexity $O(k^2)$. In contrast, the calculation of M by Omega [154] was performed in $O(k^3)$. The algorithm proceeds in two steps: initially, all pairwise LD values between all SNPs in the region are computed, and thereafter all sums of LD values between all SNP pairs are computed. Given S SNPs in a genomic region G, a matrix M of size $S^2/2$ is computed as follows:

$$M_{i,j} = \begin{cases} 0 & 1 \leq i \leq S,\ j = i \\ r_{ij}^2 & 2 \leq i \leq S,\ j = i-1 \\ M_{i,j+1} + M_{i-1,j} + & \\ M_{i-1,j+1} + r_{ij}^2 & 3 \leq i < S,\ i-1 > j \geq 0. \end{cases} \tag{8.2}$$

8.3.5 Reuse of LD values

To accelerate computations, we reuse the linkage-disequilibrium values that have already been calculated for previous grid positions in the case that there is an overlap with the current position. In general, the overlap between genomic regions depends on the size of the window defined by the user, the length of the alignment, and the number N of distinct ω positions along the alignment. Prior to calculating the LD values that correspond to position i, OmegaPlus compares the region borders of grid positions i and $i-1$. If the regions overlap, the previously calculated LD values for position $i-1$ in M are re-indexed such that they can be reused to calculate LD values at position i. The performance improvement induced by this optimization depends on the overlap between neighboring genomic regions. Typically, re-using LD values reduces execution times by up to one order of magnitude.

8.4 Parallel implementations

Figure 8.4 shows the basic steps for the calculation of an ω value at a given position. The same procedure is repeated to compute ω values at all positions specified by the `-grid` option.

8.4.1 Fine-grain parallelism in OmegaPlus-F

The fine-grain parallel version of OmegaPlus exploits parallelism within the calculations of the LD values and the ω values. The fine-grain parallelization of Equation 8.2 is implemented by evenly distributing the independent r_{ij}^2 calculations (that dominate runtimes for computing M) among all threads. Furthermore, the computation of all ω statistic values at an alignment position (one ω statistic value for every l in Equation 8.1) is also evenly distributed among threads. Figure 8.5 illustrates how the basic algorithmic steps of Figure 8.4 are executed by multiple threads using this fine-grain parallelization scheme. As already mentioned in Section 8.1, the computation of the r_{ij}^2 values for a small number of sequences is fast relative to thread synchronization times. Furthermore, if the probability of more than one mutations occuring at the same site is zero (infinite site model [107]), the DNA input data can be transformed into a binary data representation (derived/ancestral states). In this case, r_{ij}^2 computations become even faster because computing r_{ij}^2 on DNA data requires approximately 10 times (on average) more arithmetic operations than on binary data.

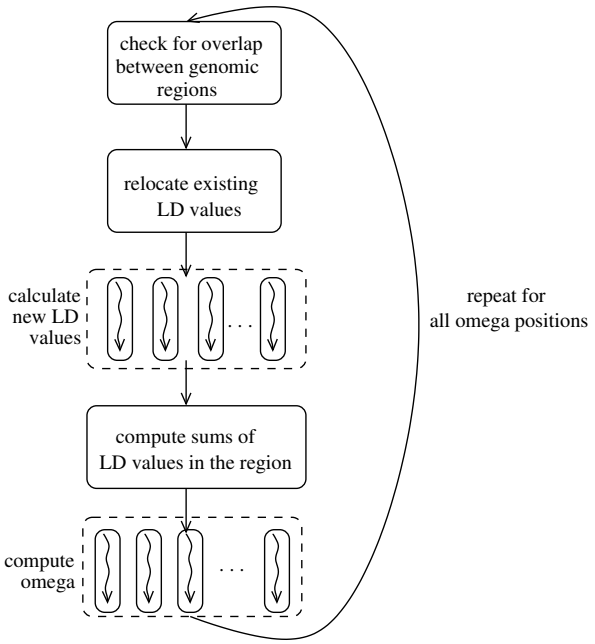


Figure 8.5: Fine-grain OmegaPlus parallelization.

8.4.2 Coarse-grain parallelism in OmegaPlus-C

The coarse-grain version assigns the grid positions to different threads. Unlike the fine-grain approach, the coarse-grain scheme is not affected by an unfavorable computation-to-synchronization ratio since essentially no thread synchronization is required. Each thread is assigned a different sub-genomic region (part of the input MSA) and carries out all ω statistic calculations in that region. Note that we generate as many sub-genomic regions (tasks) as there are threads/cores available. Thus, threads need to synchronize only once, to determine if they have all completed the analysis of their individual sub-genomic region/task. Figure 8.6 outlines this coarse-grain parallelization scheme. Since synchronization is only required to ensure that all tasks have been completed, the performance of the coarse-grain approach is limited by the runtime of the slowest thread/task. The slowest thread is always the one that has been assigned to the sub-genomic region with the highest number of SNPs.

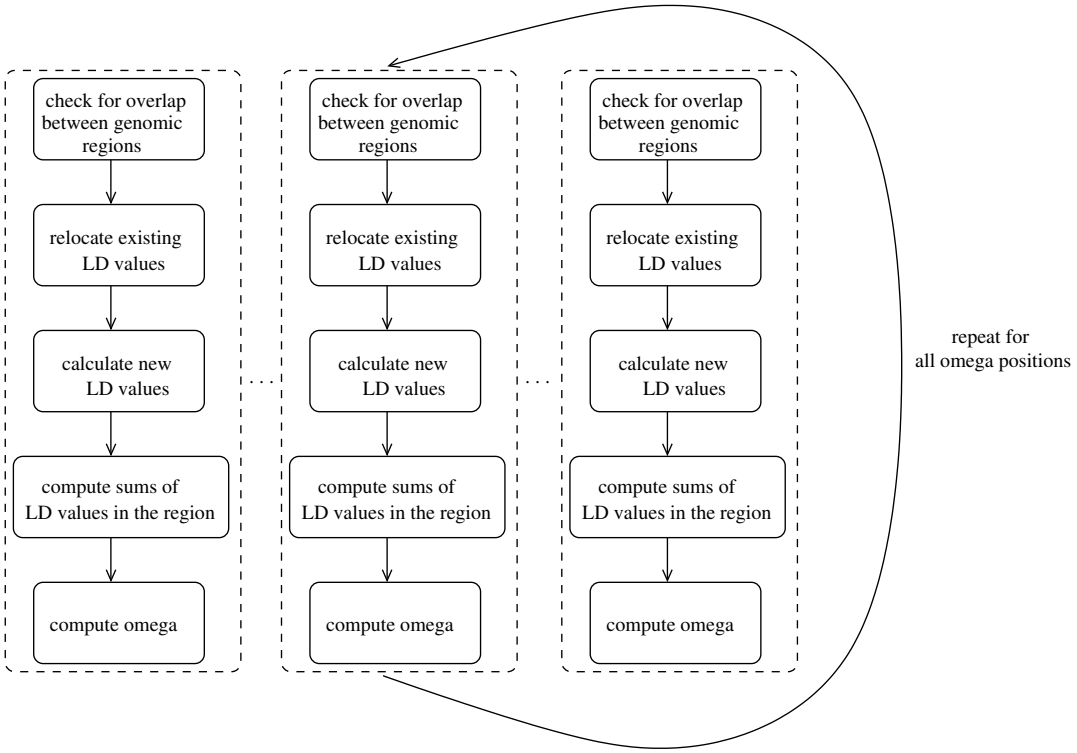


Figure 8.6: Coarse-grain OmegaPlus parallelization.

8.4.3 Multi-grain parallelism in OmegaPlus-M

To alleviate the load imbalance caused by regions of high SNP density, we introduce a multi-grain parallelization scheme for reducing the runtime of the slowest threads/tasks by using a hybrid coarse-grain/fine-grain strategy.

Figure 8.7 outlines the possible operational states of a thread during the analysis of a MSA when the multi-grain approach is used. Initially, the approach is similar to the coarse-grain scheme: each thread is assigned a sub-genomic region of equal size (but potentially different density) and carries out the ω statistic operations within that region independently and sequentially. However, when a thread finishes processing its task/region in the coarse-grain approach, it starts executing a busy-wait until the last thread has completed its task. As already mentioned, the threads are only synchronized once via such a busy-wait barrier in the very end. In the multi-grain implementation, when a thread finishes processing its own task, it notifies the other threads which are still working on their tasks that it is available to help accelerate computations. At each point in time, only the slowest among all threads will obtain help from all other available threads that have completed their tasks. We use the term *temporary workers* to refer to the threads that are available to help others, and the term *temporary master* to refer to the thread that is getting help from the *temporary workers*.

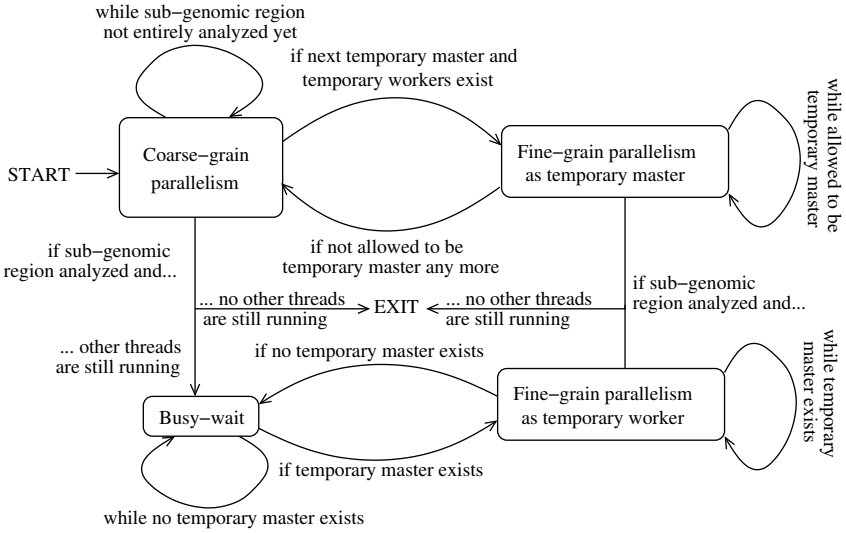


Figure 8.7: Processing states of a thread according to the multi-grain parallel model.

The *temporary master* is responsible to assign temporary thread IDs to the *temporary workers*, synchronize them, and release them after a certain period of time. The *temporary master* can not use the *temporary workers* for an unlimited amount of time, since another thread will become the slowest thread at some later point in time because of the speedup in computations attained by using the *temporary workers*. Only the current *temporary master* is allowed to designate the next *temporary master*. This approach is used to prevent slow threads from competing for *temporary workers* and also to avoid excessive occupation of the *temporary workers* by the same *temporary master*. Every *temporary master* uses the workers for the calculation of the ω statistic at 25 positions. This number has been determined experimentally. Smaller numbers led to increased synchronization overhead between threads, while larger numbers led

to increased imbalance between the slowest threads. While the *temporary master* is using the *temporary workers* via fine-grain parallelism, the other threads (on remaining regions) are still operating in coarse-grain mode. When a *temporary master* renounces its privilege to use the *temporary workers*, it returns to coarse-grain parallel mode.

The multi-grain parallelization and synchronization is implemented via integer arrays (*available* array, *turn* array, *progress* array). The *available* array shows which threads have completed their task and are ready to help others with computations. The *turn* array indicates which thread is allowed to use the *temporary workers* at each point in time. Finally, the *progress* array shows the progress each thread has made with the analysis of its sub-genomic region. Progress is measured as the number of L_i positions ($1 < i < N$; N is defined by the user, see Section 8.2) in the sub-genomic region of the thread at which the ω statistic has been calculated.

At a given point in time, the slowest thread is determined by searching for the highest number of still uncalculated L_i positions in the *progress* array. The *temporary master* scans the *progress* array and announces which thread shall become the next *temporary master* by updating the *turn* array. After each ω computation at an L_i position by every unfinished thread, all threads check their corresponding entry in the *turn* array to find out whether it is their turn to utilize the *temporary workers*. The thread that is allowed to utilize the *temporary workers* acquires them by marking them as unavailable in the *available* array. Then, temporary thread IDs are assigned by the *temporary master* to all available *temporary workers* via the *worker_ID* array. The *temporary master* also assigns a temporary thread ID to itself. The temporary thread ID assignment guarantees correct indexing and work distribution in fine-grain parallel computations of the M matrix. For the *temporary master*, the temporary ID also corresponds to the total number of parallel threads that will be used to accelerate computations. The *temporary master* informs the *temporary workers* about the thread ID of the *temporary master* via the *master_ID* array. To ensure consistency, it is important for the *temporary workers* to know the ID of their *temporary master* such as to avoid storing L_i values that correspond to the sub-genomic region of the *temporary master* in memory space that has been allocated by other threads that are still running.

8.5 Usage example

To demonstrate the ability to efficiently process whole-genome datasets, we used an off-the-shelf laptop with an Intel Core i5 CPU running at 2.53 GHz and 4 GBs RAM (Ubuntu 10.04.3). The X chromosome of 37 *Drosophila melanogaster* genomes sampled in Raleigh, North Carolina, was analyzed. The sequences (available from the Drosophila Population Genomics Project at `www.dpgp.org`) were converted from fastq to fasta using a script (available at `www.dpgp.org`) and a Solexa quality score threshold of 30. OmegaPlus was called as follows:

```
./OmegaPlus-F -maxwin 100000 -minwin 1000 -name DPGP_chrX
              -input DPGP_chrX.fasta -grid 10000 -threads 4.
```

We calculated the ω statistic at 10,000 equidistant positions, assuming that the maximum length of L and R sub-regions to the left and right of a beneficial mutation is 100,000 base pairs. Thus, a selective sweep may affect at most 200,000 base pairs in total. The minimum length of L and R is set to 1,000 base pairs. A few entries of the report file are shown below:

```
1077252 1.640796
1079493 48.282642
1081734 2.086421.
```

Each row denotes an alignment position p and the OmegaPlus score. In this example, there is a peak at position 1,079,493.

The alignment comprised 37 sequences and 22,422,827 sites (339,710 SNPs). To analyze such an alignment, OmegaPlus used less than 2% of the available 4 GBs of main memory on average. Note that a peak memory requirement of 20.8% was recorded for some seconds to temporarily store the alignment in memory, discard non-polymorphic sites, and compress the remaining SNPs. Using both physical cores on the laptop and hyper-threading (two more virtual cores), the fine-grain version of OmegaPlus required 15 minutes. Assuming that a selective sweep only affects at most 60,000 base pairs (`-maxwin 30000`) instead of 100,000, the analysis only takes 5 minutes.

8.6 Performance evaluation

A detailed accuracy evaluation of the ω statistic as well as a comparison to alternative methods and tools such as SweepFinder [139] has already been presented in [154]. To evaluate performance of the parallel implementations, we used an AMD Opteron 6174 12-core Magny-Cours processor running at 2.2 GHz. Initially, we generated a simulated DNA dataset to assess performance of the straightforward fine-grain and coarse-grain parallel implementations. The dataset was created using Hudson's ms [91] and seq-gen [157]. A coalescent tree for 1,000 chromosomes was simulated with ms (ms 1000 1 -T). The coalescent tree was then passed to seq-gen to generate DNA sequences of 500,000 base pairs assuming the HKY [85] nucleotide substitution model and branch scaling parameter 0.1 (`seq-gen -mHKY -s0.1 -l500000`). The simulated dataset contained 226,625 SNPs. We calculated the ω statistic at 10 positions assuming that the maximum region a sweep might have affected is 20,000 base pairs:

```
./OmegaPlus -grid 10 -maxwin 10000 -minwin 100 -name DNA1000.RUN
            -input DNA1000.fasta.
```

All parallel implementations comprise an initial sequential part. This is the code that reads the input data from the file and discards the non-polymorphic sites. The times for the compute-intensive part of the runs (ω statistic computations) are shown in Table 8.1. The table also shows execution times when the same dataset is converted to a binary representation (for performance reasons) using the *-binary* optimization flag. The sequential part of all DNA runs took between

46.1 and 53.5 seconds, while for binary runs (including the conversion to binary representation) it required between 61.4 and 70.0 seconds. The computational part of the original OmegaPlus code required 1,112.1 seconds to process the DNA data and 119.6 seconds to process the same data in binary representation. Figure 8.8 illustrates the speedups for the computational parts of the programs when 2, 4, 6, 8, 10, and 12 threads are used.

Threads	OmegaPlus-F (binary)	OmegaPlus-C (binary)	OmegaPlus-F (DNA)	OmegaPlus-C (DNA)
2	70.9	60.3	561.1	465.3
4	40.9	46.0	286.5	424.7
6	30.2	32.6	194.7	290.1
8	26.0	33.3	148.6	288.7
10	22.6	19.6	120.2	165.4
12	22.8	18.7	111.7	170.1

Table 8.1: Execution times (only for the ω statistic computations) of OmegaPlus, OmegaPlus-F, and OmegaPlus-C to process an alignment of 1,000 sequences and 500,000 sites.

OmegaPlus-F requires frequent synchronizations between threads for the calculation of every ω. The threads are synchronized twice: once after the LD calculations and once after the ω calculations. The amount of arithmetic operations required to calculate the LD values on binary data is significantly smaller than that for DNA data (approximately 10 times less operations on average). This means that the amount of work the threads have to execute in-between synchronization points is smaller for binary data, which deteriorates performance because of an unfavorable computation-to-synchronization ratio. Note that we already use a very efficient thread synchronization mechanism that relies on a busy-wait strategy [34]. In terms of memory consumption, OmegaPlus and OmegaPlus-F require exactly the same amount of memory to analyze an alignment. On the other hand, OmegaPlus-C requires more memory because each thread has to allocate and operate on a private LD matrix space. Thus, the memory consumption of the coarse-grain implementation is proportional to the number of threads.

To evaluate performance of the multi-grain parallelization, we performed experiments using both real-world (the 37 *Drosophila melanogaster* genomes) as well as simulated datasets, and compared execution times among all three parallelization schemes. Once again, we used Hudson's ms to generate simulated data, but this time assuming past population size changes (population bottleneck [73]) and positive recombination probability between base pairs. Simulating population bottlenecks in conjunction with recombination events increases the variance between genomic regions [82], thus leading to higher variability in SNP density. The simulated alignment was generated with the following command:

```
ms 1000 1 -r 50000 1000 -eN 0.001 0.001 -eN 0.002 10 -s 10000.
```

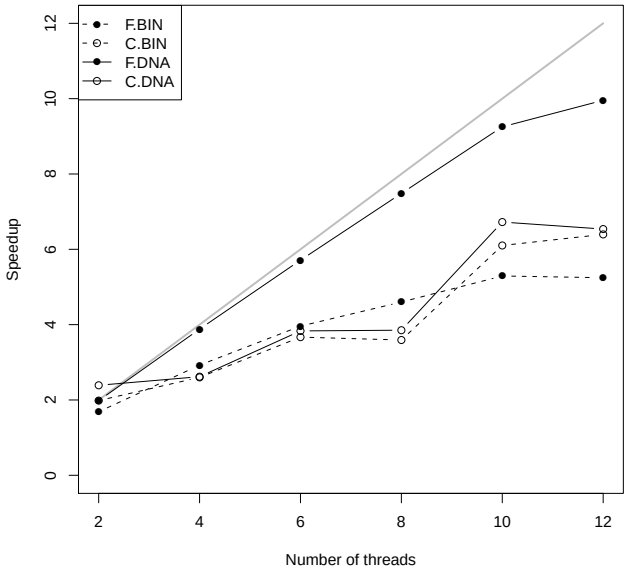


Figure 8.8: Acceleration of the ω statistic computations using fine-grain (F) and coarse-grain (C) parallelism on DNA and binary (BIN) data. The gray line illustrates linear speedup.

Flags `-eN 0.001 0.001` and `-eN 0.002 10` specify a population bottleneck. Backward in time (from present to past), the population has contracted to the 0.001 of its present-day size at time 0.001. Then, at time 0.002, the population became 10 times larger than the present-day population. Time is measured in $4N$ generations [91], where N is the effective population size. Regarding recombination (`-r 50000 1000`), we assume that there are 1,000 potential breakpoints where recombination may have occured and that the total rate of recombination is 50,000 (i.e., $4Nr = 50,000$, where r is the recombination rate between two adjacent sites in the alignment and N is the effective population size).

Table 8.2 provides the total execution times required by OmegaPlus-F (fine-grain), OmegaPlus-C (coarse-grain), and OmegaPlus-M (multi-grain) using 2 up to 12 cores for analyzing an average-size alignment with 1,000 sequences and 100,000 alignment sites comprising 10,000 SNPs. All OmegaPlus runs calculated the ω statistic at 10,000 positions along the alignment assuming that the maximum size of the neighborhood of a beneficial mutation that a selective sweep might have affected is 20,000 alignment sites. The sequential version of the code (OmegaPlus) required 317 seconds to process this alignment on one core of the test platform.

The last two rows of Table 8.2 show the performance improvement over the fine- and coarse-grain parallel implementations for different numbers of threads. On this simulated dataset, the multi-grain approach is between 6.3% (using two threads) and 35.9% (using 12 threads) faster than the fine-grain approach, and between 16.8% (using two threads) and 39.0% (using

Parallelization approach	Number of threads					
	2	4	6	8	10	12
Fine-grain	211.1	156.6	126.0	113.3	107.5	99.5
Coarse-grain	237.8	214.4	164.5	110.7	116.1	101.1
Multi-grain	197.8	135.0	100.3	87.2	77.1	63.8
Multi vs Fine (%)	6.3	13.7	20.3	23.0	28.2	35.9
Multi vs Coarse (%)	16.8	37.0	39.0	21.2	33.6	36.9

Table 8.2: Total execution times (in seconds) of OmegaPlus-F, OmegaPlus-C, and OmegaPlus-M to analyze an alignment of 1,000 sequences and 100,000 sites (10,000 SNPs). The last two rows show the performance improvement of the multi-grain version over the fine-grain (Multi vs Fine) and the coarse-grain (Multi vs Coarse) implementations.

6 threads) faster than the coarse-grain approach.

The improvement differences among runs with different number of threads are caused by changes in the size of the sub-genomic regions when coarse-grain parallelization is used. Since there are as many sub-genomic regions as threads available, the number of threads affects the size of these regions and as a consequence the variability of SNP density among them. The increasing parallel efficiency of OmegaPlus-M with increasing number of threads compared to OmegaPlus-F can be attributed to the effectiveness of the initial coarse-grain parallel operations on binary data. As already mentioned, ω statistic computations on binary data require 10 times less operations (on average) than on DNA data. Therefore, when the fine-grain parallel scheme is used to analyze binary data, the synchronization-to-computation ratio is high in comparison to a DNA data analysis. As a consequence, the coarse-grain parallelization is more efficient for analyzing binary data than the fine-grain approach.

Parallelization approach	Number of threads					
	2	4	6	8	10	12
Fine-grain	1105.8	760.8	599.9	565.6	555.1	534.0
Coarse-grain	866.5	477.8	348.9	277.3	247.2	231.7
Multi-grain	857.1	437.3	312.1	248.0	210.8	190.6
Multi vs Fine (%)	22.4	42.5	47.9	56.1	62.1	64.4
Multi vs Coarse (%)	1.0	8.3	10.3	10.4	14.9	17.7

Table 8.3: Total execution times (in seconds) of OmegaPlus-F, OmegaPlus-C, and OmegaPlus-M to analyze the real-world *Drosophila melanogaster* dataset (37 sequences and 339,710 SNPs in 22,422,827 alignment sites).

Table 8.3 shows the results of the runs on the real-world *Drosophila melanogaster* MSA. Once again, we calculated the ω statistic at 10,000 positions along the alignment, but this

time using a maximum neighborhood size of 200,000 alignment sites. On a single core of our test platform, the sequential version required 1,599.8 seconds. Once again, OmegaPlus-M outperformed OmegaPlus-F and OmegaPlus-C in all runs (each run with a different number of threads). OmegaPlus-M yielded parallel performance improvements ranging between 22.4% and 64.4% over OmegaPlus-F and between 1.0% and 17.7% over OmegaPlus-C.

Unlike the results on the simulated dataset, performance continuously improves on the real dataset as we use more threads. This is because the actual number of SNPs in the *Drosophila melanogaster* MSA is one order of magnitude larger than the number of SNPs in the simulated dataset. Despite the fact that the number of ω statistic positions per region is the same in both MSAs, the ω statistic computations on the real-world dataset comprise approximately 45,000 to 60,000 SNPs in each region, whereas there are only 2,500 to 6,000 SNPs in each region of the simulated dataset. A detailed analysis of the computational load per region revealed that the size of the dynamic programming matrices when the *Drosophila melanogaster* DNA MSA (average number of cells: 4,859,088) is analyzed is twice the size of the respective matrices for the analysis of the simulated binary dataset (average number of cells: 2,239,508). The load analysis also revealed that the overlap between neighborhoods around consecutive ω statistic positions is higher for the simulated dataset. As a consequence, a significantly higher amount of operations is required per matrix for the *Drosophila melanogaster* MSA, thereby improving the synchronization-to-computation ratio in the fine-grain scheme and allowing for better scalability.

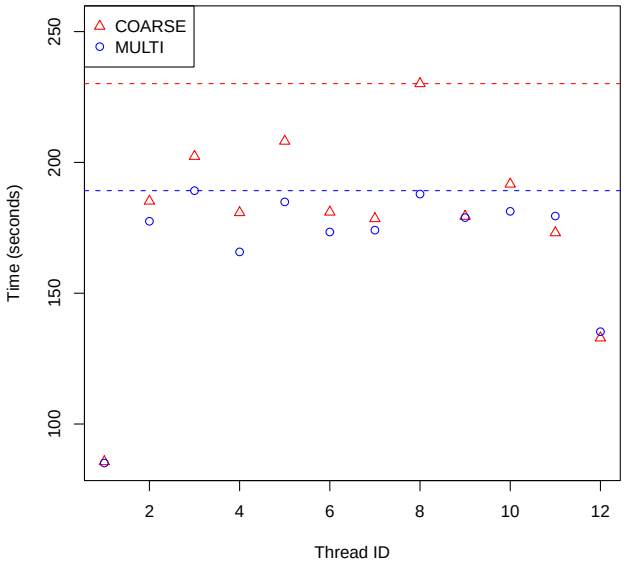


Figure 8.9: Per-thread execution times (in seconds) of OmegaPlus-C and OmegaPlus-M when the *Drosophila melanogaster* MSA is analyzed.

The multi-grain approach aims at improving the parallel efficiency of the coarse-grain approach. This is achieved by reducing the runtime of the slowest threads/tasks by means of fine-grain parallelism. Figure 8.9 depicts the effect of the multi-grain parallelization on the execution times of the individual threads/tasks when 12 threads are used to analyze the *Drosophila melanogaster* MSA. The red and blue horizontal lines show the points in time when the slowest threads in OmegaPlus-C and OmegaPlus-M terminate, respectively.

8.7 Summary

This chapter presented OmegaPlus, an efficient implementation of ω statistic computations to accurately identify selective sweeps in whole-genome data based on linkage-disequilibrium patterns. Existing tools for this purpose are not suitable for genome-size analyses because of excessive memory requirements or due to algorithmic restrictions that only allow for analyzing sub-genomic regions.

To process input alignments, OmegaPlus adopts a window-based approach that allows for analyses of long, genome-size MSAs, while the memory requirements remain independent of the alignment size. Furthermore, OmegaPlus introduces a novel dynamic programming algorithm for the computation of sums of linkage-disequilibrium values in every alignment window. This approach reduces the problem complexity and significantly accelerates computations.

In addition to the algorithmic advances, three parallelization schemes have been developed and implemented to allow for efficient exploitation of all cores in multi-core CPUs. Fine-grain parallelism evenly distributes LD and ω statistic computations to all threads. On the other hand, coarse-grain parallelism organizes neighboring sub-genomic regions into groups and assigns each of these groups to a different thread that carries out all computations sequentially. Finally, an efficient multi-grain parallelization scheme has been devised to better distribute load among threads. The underlying idea of the multi-grain approach is to use fast threads (that finish their task before others in a coarse-grain model) to help slower threads in accomplishing their tasks by deploying fine-grain parallelism.

The latest version of OmegaPlus (version 2.0.0) analyzed the X chromosome of 37 *Drosophila melanogaster* genomes (available from the Drosophila Population Genomics Project at `www.dpgp.org`) in less than 5 minutes using only 2% of main memory on an off-the-shelf personal laptop (Intel Core i5 CPU with 4 GBs RAM running at 2.53 GHz). Employing multi-threading and algorithmic optimizations (e.g., conversion to binary representation), the analysis of the DPGP dataset required less than a minute. This example shows the capacity of OmegaPlus to rapidly analyze whole-genome datasets without excessive computational and/or memory requirements.

Chapter 9

Conclusion and Future Work

This chapter provides a conclusion and summary of the work conducted and addresses some possible directions of future work.

9.1 Conclusion

Real-world phylogenetic and population genetic analyses are characterized by excessive runtimes. This is due to the computational requirements of the employed kernels and/or the large number of kernel invocations that is required for the analyses. This justifies the need for accelerating and/or re-designing kernels to make them run more efficiently, and thus boost overall performance of the applications.

This thesis focused on the analysis of the computational kernels of important evolutionary Bioinformatics methods. We designed and developed dedicated computer architectures for these kernels as well as software-based parallelized solutions. Chapters 4 and 5 presented reconfigurable architectures for the acceleration of the phylogenetic parsimony and likelihood functions that can be used to score alternative tree topologies for phylogenetic inference. Chapter 6 focused on the acceleration of a newly introduced phylogeny-aware short DNA read alignment kernel. The chapter presented a complete hardware-based co-processor approach for offloading the alignment kernel onto an FPGA, an efficient vectorization of the kernel for modern x86 CPU architectures, and a series of effective optimizations for porting the kernel to GPUs. Finally, a load distribution mechanism was devised for a hybrid CPU-GPU approach that can exploit all computational resources on modern workstations.

Apart from designing application-specific architectures, this thesis also introduced novel algorithmic approaches for specific phylogenetic and population genetic problems. Chapter 7 presented a set of rules for conducting efficient tree searches on phylogenomic alignments with missing data. These rules allow for extracting individual per-gene subtrees from a comprehensive tree based on the pattern of missing per-gene data, and to maintain these subtrees in a consistent state while conducting tree searches on the global tree. Using this technique, the memory requirements and number of floating-point operations are reduced in proportion to

the amount of missing data in the alignment. Finally, Chapter 8 introduced OmegaPlus, an algorithmically enhanced implementation of the ω statistic that is used in population genetic analyses to detect recent and strong positive selection. The algorithmic advances in combination with the parallelization allow for the rapid and memory-efficient analysis of genome-size datasets.

Furthermore, for all software- and hardware-based solutions presented in this thesis, a significant effort was undertaken to carry out as fair as possible performance evaluations and comparisons. Previous work on the acceleration of Bioinformatics kernels reported performance differences between FPGAs, GPUs, and CPUs in the range of two to three orders of magnitude. This thesis revealed that, when performance evaluations are based on thoroughly optimized code for x86 architectures, state-of-the-art multi-core CPUs deliver comparable performance to state-of-the-art GPUs. Moreover, dedicated architectures mapped on new-generation FPGAs can only be up to 10 times faster.

Despite the fact that all software- and hardware-based solutions presented in this thesis have been verified for correctness and evaluated with respect to their performance, there are several approaches to further improve performance, which are addressed in the following section.

9.2 Future work

A recurrent problem in co-processor design is the data movement from and to the memory blocks of the co-processor. Regardless of the underlying technology, the interconnect between a host microprocessor architecture and a dedicated co-processor has to be fast enough such that data transfers do not hinder the co-processor from delivering peak performance. An Ethernet-based communication platform was developed to facilitate the verification and testing process of the reconfigurable architectures presented in this thesis, but even faster solutions, such as PCI Express, are required in order to reduce the impact of data transfers on overall system performance. Furthermore, for certain applications, it is not unusual to exhibit input/output requirements that PCI Express can not meet, mainly because of the ability of FPGAs to carry out a large number of operations in parallel. The phylogenetic likelihood function architectures (presented in Chapter 5) for instance, require complex data transfer systems that rely on FPGA on-chip memory blocks for overlapping communication with computations.

For large-scale analyses, an issue that usually arises is that the amount of data to be processed does not fit in on-chip memory blocks nor in on-board DRAM memory. In such cases, continuous data transfers between the host memory and the co-processor are required. Evidently, a hierarchical memory subsystem is required for these cases. To ensure that such continuous data transfers will not deteriorate overall system performance, the co-processor needs to utilize the on-chip memory as L1 cache while the on-board DRAM memory serves as L2 cache. Since the co-processor can not and should not have access to the memory on the host system, the design of such a hierarchical cache-like data transfer system represents an open

software/hardware co-design challenge.

Another aspect of future work that needs to be investigated is the problem of load imbalance. The design of hybrid systems that typically employ one or more CPUs and one or more co-processors (GPUs, FPGAs, etc.) requires an optimal offloading of the computational tasks to the available resources. Here, this has only been done for the GPU-based acceleration of the phylogeny-aware read alignment kernel described in Chapter 7. Similar techniques need to be devised for the phylogenetic parsimony and likelihood architectures for using them in real-world analyses.

The thread scheduling problem arises when multiple threads are deployed to accomplish one or more tasks in software-only implementations. Chapter 8 presented a promising multi-grain parallelization technique for the OmegaPlus code. The underlying idea is to assign all available threads to accelerate the thread which will most likely terminate last at the point in time that the available threads are allocated to help that thread. What still remains to be explored is how to optimally distribute the fast threads to the slowest one(s), potentially based on a pre-analysis of the variability of SNP density along the MSA in combination with a detailed empirical investigation of thread behavior as a function of the assigned alignment region. This will allow to further optimize the assignment of worker thread(s) to master thread(s).

Finally, our long-term vision is to design a generic Bioinformatics processor architecture that may serve as reference for a future ASIC. The analysis of the computational requirements of some representative kernels in the fields of phylogenetics and population genetics provided insights into the characteristics of the most commonly used operations. For example, matrix-vector multiplications are required for the likelihood function, and population count operations are required for the parsimony function and the ω statistic. Such requirement analyses may be regarded as the first step toward the design of optimized ALUs that are adapted to the needs of Bioinformatics applications. Extending such Bioinformatics-specific ALUs through instruction fetch and decode units, as well as a register file to create a pipelined architecture that operates in a RISC-like fashion, will enable programmers to embed assembly code into widely used Bioinformatics tools in an analogous way as vector instructions are used today.

Bibliography

[1] Advanced Micro Devices, Inc. http://www.amd.com.

[2] Altera Corporation. http://www.altera.com.

[3] Cadense Design Systems, Inc. http://www.cadence.com/us/pages/default.aspx.

[4] GARLI download page. http://garli.googlecode.com.

[5] NVIDIA Corporation. http://www.nvidia.com.

[6] OmegaPlus 2.0.0 Manual. http://www.exelixis-lab.org/omegaManual.php.

[7] Phylogeny Programs. http://evolution.genetics.washington.edu/phylip/software.html.

[8] Xilinx Inc. http://www.xilinx.com.

[9] J. Adachi and M. Hasegawa. *MOLPHY, programs for molecular phylogenetics, I: PROTML, maximum likelihood inference of protein phylogeny.* Number 27. Institute of Statistical Mathematics, 1992.

[10] N. Alachiotis, S. Berger, and A. Stamatakis. Efficient PC-FPGA Communication over Gigabit Ethernet. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1727–1734. IEEE, 2010.

[11] N. Alachiotis, S. Berger, and A. Stamatakis. Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 226–233. IEEE, 2011.

[12] N. Alachiotis, S. Berger, and A. Stamatakis. Coupling SIMD and SIMT Architectures to Boost Performance of a Phylogeny-aware Alignment Kernel. Submitted to BMC Bioinformatics.

[13] N. Alachiotis, P. Pavlidis, and A. Stamatakis. Exploiting Multi-grain Parallelism for efficient Selective Sweep Detection. ICA3PP-12 (Accepted for publication).

[14] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis. Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2009 IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[15] N. Alachiotis and A. Stamatakis. A Generic and Versatile Architecture for Inference of Evolutionary Trees under Maximum Likelihood. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pages 829–835. IEEE, 2010.

[16] N. Alachiotis and A. Stamatakis. Efficient floating-point logarithm unit for FPGAs. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[17] N. Alachiotis and A. Stamatakis. A Vector-Like Reconfigurable Floating-Point Unit for the Logarithm. *International Journal of Reconfigurable Computing*, 2011.

[18] N. Alachiotis and A. Stamatakis. FPGA Acceleration of the Phylogenetic Parsimony Kernel? In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 417–422. IEEE, 2011.

[19] N. Alachiotis and A. Stamatakis. FPGA Optimizations for a Pipelined Floating-Point Exponential Unit. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 316–327, 2011.

[20] N. Alachiotis, A. Stamatakis, and P. Pavlidis. OmegaPlus: A Scalable Tool for Rapid Detection of Selective Sweeps in Whole-Genome Datasets. Submitted to Bioinformatics.

[21] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas. A Reconfigurable Architecture for the Phylogenetic Likelihood Function. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 674–678. IEEE, 2009.

[22] B. Alpern, L. Carter, and K. Su Gatlin. Microparallelism and high-performance protein matching. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 24. ACM, 1995.

[23] G. Altekar, S. Dwarkadas, J. P. Huelsenbeck, and F. Ronquist. Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. *Bioinformatics*, 20(3):407–415, 2004.

[24] Altera Corporation. *FPGA architecture*. www.altera.com/literature/wp/wp-01003.pdf.

[25] S. Altschul and B. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of mathematical biology*, 48(5):603–616, 1986.

[26] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[27] D. Bader, B. Moret, and L. Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. In *Proc. of SPIE ITCom*, volume 4528, pages 159–168. Citeseer, 2001.

[28] D. Bader, U. Roshan, and A. Stamatakis. Computational grand challenges in assembling the tree of life: Problems and solutions. *Advances in computers*, 68:127–176, 2006.

[29] J. Bakos. FPGA Acceleration of Gene Rearrangement Analysis. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 85–94, 2007.

[30] J. Bakos, P. Elenis, and J. Tang. FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data. In *Bioinformatics and Bioengineering, 2007. BIBE 2007. Proceedings of the 7th IEEE International Conference on*, pages 888–895, 2007.

[31] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, et al. Maxwell-a 64 FPGA supercomputer. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 287–294. IEEE, 2007.

[32] S. Berger, N. Alachiotis, and A. Stamatakis. An Optimized Reconfigurable System for Computing the Phylogenetic Likelihood on DNA Data. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2009 IEEE International Symposium on*, pages 1–8. IEEE, 2012.

[33] S. Berger and A. Stamatakis. Accuracy and Performance of Single versus Double Precision Arithmetics for Maximum Likelihood Phylogeny Reconstruction. In *Proceedings of the Parallel Biocomputing Workshop 2009 (PBC09)*, 2009.

[34] S. Berger and A. Stamatakis. Assessment of barrier implementations for fine-grain parallel regions on current multi-core architectures. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[35] S. Berger and A. Stamatakis. Aligning short reads to reference alignments and trees. *Bioinformatics*, 27(15):2068–2075, 2011.

[36] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, and C. Antonopoulos. Dynamic multigrain parallelization on the Cell Broadband Engine. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–100. ACM, 2007.

[37] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos. RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[38] R. K. Bradley, A. Roberts, M. Smoot, S. Juvekar, J. Do, C. Dewey, I. Holmes, and L. Pachter. Fast Statistical Alignment. *PLoS Comput Biol*, 5(5):e1000392, 2009.

[39] J. Braverman, R. Hudson, N. Kaplan, C. Langley, and W. Stephan. The hitchhiking effect on the site frequency spectrum of DNA polymorphisms. *Genetics*, 140(2):783–796, 1995.

[40] R. Brent. *Algorithms for minimization without derivatives*. Dover Pubns, 2002.

[41] L. D. Bromham, A. E. Rambaut, and P. H. Harvey. Determinants of rate variation in DNA sequence evolution of mammals. *Molecular Evolution*, 43:610–621, 1996.

[42] J. Brown and P. Warren. Antibiotic discovery: Is it in the genes? *Drug Discovery Today*, 3:564–566, 1998.

[43] R. M. Bush, C. A. Bender, K. Subbarao, N. J. Cox, and W. M. Fitch. Predicting the Evolution of Human Influenza A. *Science*, 286(5446):1921–1925, 1999.

[44] G. K. Chen, P. Marjoram, and J. D. Wall. Fast and flexible simulation of DNA sequence data. *Genome Res*, 19(1):136–142, 2009.

[45] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation—a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[46] B. Chor and T. Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21(1):97–106, 2005.

[47] K. Compton and S. Hauck. An introduction to reconfigurable computing. *IEEE Computer*, 2000.

[48] J. F. Crow and M. Kimura. *An introduction to population genetics theory.* Harper & Row New York, 1970.

[49] J. Davis, S. Akella, and P. Waddell. Accelerating phylogenetics computing on the desktop: experiments with executing UPGMA in programmable logic. In *Engineering in Medicine and Biology Society, 2004. IEMBS'04. 26th Annual International Conference of the IEEE*, volume 2, pages 2864–2868. IEEE, 2004.

[50] W. H. E. Day, D. S. Johnson, and D. Sankoff. The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical Biosciences*, 81:33–42, 1986.

[51] F. De Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 59–64. IEEE, 2009.

[52] C. Dunn, A. Hejnol, D. Matus, K. Pang, W. Browne, S. Smith, E. Seaver, G. Rouse, M. Obst, G. Edgecombe, M. Sorensen, S. Haddock, A. Schmidt-Rhaesa, A. Okusu, R. Kristensen, W. Wheeler, M. Martindale, and G. Giribet. Broad phylogenomic sampling improves resolution of the animal tree of life. *Nature*, 452(7188):745–749, 2008.

[53] S. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[54] R. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[55] R. Edgar and S. Batzoglou. Multiple sequence alignment. *Current opinion in structural biology*, 16(3):368–373, 2006.

[56] A. W. F. Edwards and L. L. Cavalli-Sforza. Reconstruction of evolutionary trees. In V. H. Heywood and J. McNeill, editors, *Phenetic and Phylogenetic Classification*, volume 6 of *Systematics Association*, pages 67–76. Systematics Association, London, 1964.

[57] B. Efron. Bootstrap methods: Another look at the Jackknife. *The annals of Statistics*, 7(1):1–26, 1979.

[58] J. A. Eisen. Phylogenomics: Improving Functional Predictions for Uncharacterized Genes by Evolutionary Analysis. *Genome Research*, 8(3):163–167, 1998.

[59] R. Elston and J. Stewart. A general model for the genetic analysis of pedigree data. *Human heredity*, 21(6):523–542, 1971.

[60] B. Engelen, K. Meinken, F. von Wintzingerode, H. Heuer, H.-P. Malkomes, and H. Backhaus. Monitoring Impact of a Pesticide Treatment on Bacterial Soil Communities by Metabolic and Genetic Fingerprinting in Addition to Conventional Testing Procedures. *Appl. Environ. Microbiol.*, 64(8):2814–2821, 1998.

[61] J. Eusse Giraldo, N. Moreano, R. Jacobi, and A. de Melo. A HMMER hardware accelerator using divergences. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 405–410. European Design and Automation Association, 2010.

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

[63] J. Felsenstein. *Statistical Inference and the Estimation of Phylogenies*. PhD thesis, Univ. Chicago, Chicago, 1968.

[64] J. Felsenstein. Cases in which parsimony or compatibility methods will be positively misleading. *Systematic Biology*, 27(4):401–410, 1978.

[65] J. Felsenstein. Evolutionary Trees from DNA Sequences: A Maximum Likelihood Approach. *Molecular Evolution*, 17:368–376, 1981.

[66] J. Felsenstein. Confidence limits on phylogenies: an approach using the bootstrap. *Evolution*, pages 783–791, 1985.

[67] J. Felsenstein. *PHYLIP (Phylogeny Inference Package) version 3.69*. Department of Genome Sciences and Department of Biology, University of Washington, Seattle, Washington, 2009. Distributed by the author.

[68] W. M. Fitch. Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, 20(4):406–416, 1971.

[69] R. Fletcher. *Practical methods of optimization, Volume 1*. Wiley, 1987.

[70] B. S. Gaut, S. V. Muse, W. D. Clark, and M. T. Clegg. Relative rates of nucleotide substitution at the rbcL locus of monocotyledonous plants. *Molecular Evolution*, 35:292–303, 1992.

[71] C. J. Geyer. Markov chain Monte Carlo maximum likelihood. In E. M. Keramides, editor, *Proceedings of the 23rd Symposium on the Interface*, Computing Science and Statistics, pages 156–163, Fairfax Station, 1991. Interface Foundation.

[72] P. Gill, W. Murray, and M. Wright. *Practical optimization*, volume 1. Academic press, 1981.

[73] J. H. Gillespie. *Population Genetics: A Concise Guide*. The Johns Hopkins University Press, 2004.

[74] P. Goloboff. Analyzing large data sets in reasonable times: solutions for composite optima. *Cladistics*, 15(4):415–428, 1999.

[75] P. Goloboff, J. Farris, and K. Nixon. TNT, a free program for phylogenetic analysis. *Cladistics*, 24(5):774–786, 2008.

[76] P. A. Goloboff, S. A. Catalano, J. M. Mirande, C. A. Szumik, J. S. Arias, M. Källersjö, and J. S. Farris. Phylogenetic analysis of 73060 taxa corroborates major eukaryotic groups. *Cladistics*, 25:1–20, 2009.

[77] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[78] O. Gotoh. Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments. *Molecular Biology*, 264(4):823–838, 1996.

[79] P. J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82:711–732, 1995.

[80] S. Guindon and O. Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic biology*, 52(5):696–704, 2003.

[81] S. Hackett, R. Kimball, S. Reddy, R. Bowie, E. Braun, M. Braun, J. Chojnowski, W. Cox, K. Han, J. Harshman, et al. A Phylogenomic Study of Birds Reveals Their Evolutionary History. *Science*, 320(5884):1763, 2008.

[82] P. R. Haddrill, K. R. Thornton, B. Charlesworth, and P. Andolfatto. Multilocus patterns of nucleotide variability and the demographic and selection history of drosophila melanogaster populations. *Genome Res*, 15(6):790–799, 2005.

[83] P. Halbur, M. Lum, X. Meng, I. Morozov, and P. Paul. New porcine reproductive and respiratory syndrome virus DNA and proteins encoded by open reading frames of an Iowa strain of the virus are used in vaccines against PRRSV in pigs, 1994. Patent filing WO9606619-A1.

[84] B. Harris, A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 765–769. IEEE, 2007.

[85] M. Hasegawa, H. Kishino, and T. Yano. Dating of the human-ape splitting by a molecular clock of mitochondrial dna. *Molecular Evolution*, 22:160–174, 1985.

[86] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[87] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, 1998.

[88] A. Hejnol, M. Obst, A. Stamatakis, M. Ott, G. Rouse, G. Edgecombe, P. Martinez, J. Baguna, X. Bailly, U. Jondelius, et al. Rooting the bilaterian tree with scalable phylogenomic and supercomputing tools. In *Proc. R. Soc. B*, volume 276, pages 4261–4270, 2009.

[89] J. Henaut, D. Dragomirescu, and R. Plana. FPGA based high date rate radio interfaces for aerospace wireless sensor systems. In *Systems, 2009. ICONS'09. Fourth International Conference on*, pages 173–178. IEEE, 2009.

[90] M. Hendy and D. Penny. A framework for the quantitative study of evolutionary trees. *Systematic Biology*, 38(4):297–309, 1989.

[91] R. R. Hudson. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337–338, 2002.

[92] J. Huelsenbeck. Performance of phylogenetic methods in simulation. *Systematic Biology*, 44(1):17–48, 1995.

[93] J. P. Huelsenbeck and F. Ronquist. MRBAYES. Bayesian inference of phylogeny. *Bioinformatics*, (17):754–755, 2001.

[94] R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: extension and analysis of the basic method. *Bioinformatics*, 12(2):95–107, 1996.

[95] J. D. Jensen, K. R. Thornton, C. D. Bustamante, and C. F. Aquadro. On the utility of linkage disequilibrium as a statistic for identifying targets of positive selection in nonequilibrium populations. *Genetics*, 176(4):2371–2379, 2007.

[96] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun. A Reconfigurable Accelerator for Smith–Waterman Algorithm. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 54(12):1077–1081, 2007.

[97] C. Johns and D. Brokenshire. Introduction to the Cell Broadband Engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[98] T. H. Jukes and C. R. Cantor. Evolution of protein molecules. In H. N. Munro and J. B. Allison, editors, *Mammalian protein metabolism*, pages 21–123. Academic Press, New York, 1969.

[99] J. Karow. *Survey: Illumina, SOLiD, and 454 Gain Ground in Research Labs; Most Users Mull Additional Purchases*. 2010. http://www.genomeweb.com/sequencing/survey-illumina-solid-and-454-gain-ground-research-labs-most-users-mull-addition.

[100] S. Kasap and K. Benkrid. A high performance FPGA-based core for phylogenetic analysis with Maximum Parsimony method. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 271–277. IEEE, 2009.

[101] S. Kasap and K. Benkrid. High Performance Phylogenetic Analysis With Maximum Parsimony on Reconfigurable Hardware. *VLSI Systems, IEEE Trans. on*, (99):1–13, 2010.

[102] K. Katoh, K. Kuma, H. Toh, and T. Miyata. MAFFT version 5: improvement in accuracy of multiple sequence alignment. *Nucleic acids research*, 33(2):511–518, 2005.

[103] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic acids research*, 30(14):3059–3066, 2002.

[104] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/opencl.

[105] Y. Kim and R. Nielsen. Linkage disequilibrium as a signature of selective sweeps. *Genetics*, 167(3):1513–1524, 2004.

[106] Y. Kim and W. Stephan. Detecting a local signature of genetic hitchhiking along a recombining chromosome. *Genetics*, 160(2):765–777, 2002.

[107] M. Kimura. The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations. *Genetics*, 61(4):893–903, 1969.

[108] M. Kimura. A simple method for estimating evolutionary rates of base substitutions by through comparative studies of nucleotide sequences. *Molecular Evolution*, 16:111–120, 1980.

[109] H. Kishino, T. Miyata, and M. Hasegawa. Maximum likelihood inference of protein phylogeny and the origin of chloroplasts. *Journal of Molecular Evolution*, 31(2):151–160, 1990.

[110] M. Kuhner and J. Felsenstein. A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Molecular Biology and Evolution*, 11(3):459–468, 1994.

[111] C. Lanave, G. Preparata, C. Sacone, and G. Serio. A new method for calculating evolutionary substitution rates. *Molecular Evolution*, 20(1):86–93, 1984.

[112] N. Lartillot, T. Lepage, and S. Blanquart. PhyloBayes 3: a Bayesian software package for phylogenetic reconstruction and molecular dating. *Bioinformatics*, 25(17):2286–2288, 2009.

[113] M. Leeser, S. Coric, E. Miller, H. Yu, and M. Trepanier. Parallel-beam backprojection: an FPGA implementation optimized for medical imaging. *The Journal of VLSI Signal Processing*, 39(3):295–311, 2005.

[114] A. S. Lewis and M. L. Overton. Eigenvalue optimization. *Acta Numerica*, 5:149–190, 1996.

[115] R. Lewontin. The interaction of selection and linkage. I. General considerations; heterotic models. *Genetics*, 49(1):49, 1964.

[116] R. Lewontin and K. Kojima. The evolutionary dynamics of complex polymorphisms. *Evolution*, pages 458–472, 1960.

[117] I. Li, W. Shum, and K. Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field-programmable gate array (FPGA). *BMC Bioinformatics*, 8(1):185, 2007.

[118] S. Li. *Phylogenetic Tree Construction Using Markov Chain Monte Carlo.* PhD thesis, Ohio State Univ., Columbus, 1996.

[119] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences of the United States of America*, 86(12):4412–4415, 1989.

[120] Y. Liu, D. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[121] Y. Liu, B. Schmidt, and D. Maskell. CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, 2010.

[122] T. Mak and K. Lam. Embedded computation of maximum-likelihood phylogeny inference using platform FPGA. In *Proceedings of IEEE Computational Systems Bioinformatics Conference (CSB 04)*, pages 512–514, 2004.

[123] T. Mak and K. Lam. FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation. *Lecture Notes in Computer Science*, pages 1076–1079, 2004.

[124] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[125] F. Matsen, R. Kodner, and E. Armbrust. pplacer: linear time maximum-likelihood and bayesian phylogenetic placement of sequences onto a fixed reference tree. *BMC Bioinformatics*, 11(1):538, 2010.

[126] B. Mau. *Bayesian Phylogenetic Inference via Markov Chain Monte Carlo Methods.* PhD thesis, Univ. Wisconsin, Madison, 1996.

[127] J. Maynard Smith and J. Haigh. The hitch-hiking effect of a favourable gene. *Genetical research*, 23(1):23–35, 1974.

[128] M. M. McMahon and M. J. Sanderson. Phylogenetic Supermatrix Analysis of GenBank Sequences from 2228 Papilionoid Legumes. *Systematic Biology*, 55(5):818–836, 2006.

[129] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953.

[130] S. Mirarab, N. Nguyen, and W. T. *SEPP: SATe-Enabled Phylogenetic Placement*. 2011. http://www.cs.utexas.edu/ tandy/warnow-psb2012.pdf.

[131] B. Moret, D. Bader, T. Warnow, S. Wyman, and M. Yan. GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. *Proc. Botany*, 2001.

[132] B. Moret, J. Tang, L. Wang, and T. Warnow. Steps toward accurate reconstructions of phylogenies from gene-order data. *Journal of Computer and System Sciences*, 65(3):508–525, 2002.

[133] D. W. Mount. *Bioinformatics: sequence and genome analysis*. CSHL press, 2004.

[134] W. B. Muir, R. Nichols, and J. Breuer. Phylogenetic analysis of varicella-zoster virus: evidence of intercontinental spread of genotypes and recombination. *J Virol*, 76(4):1971–1979, 2002.

[135] S. Needleman, C. Wunsch, et al. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[136] M. Nei. *Molecular evolutionary genetics*. Columbia Univ Press, 1987.

[137] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[138] J. Nickolls and W. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[139] R. Nielsen, S. Williamson, Y. Kim, M. J. Hubisz, A. G. Clark, and C. Bustamante. Genomic scans for selective sweeps using SNP data. *Genome Res*, 15(11):1566–1575, 2005.

[140] K. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15(4):407–414, 1999.

[141] C. Notredame and D. G. Higgins. SAGA: sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524, 1996.

[142] C. Notredame, D. G. Higgins, and J. Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Molecular Biology*, 302(1):205–217, 2000.

[143] NVIDIA. *NVIDIA CUDA C Programming Guide*. http://developer.download.nvidia.com /compute/DevZone/docs/html/C/doc/ CUDA_C_Programming_Guide.pdf.

[144] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[145] G. J. Olsen, S. Pracht, and R. Overbeek. DNArates. unpublished.

[146] M. Ott, J. Zola, S. Aluru, A. D. Johnson, D. Janies, and A. Stamatakis. Large-scale Phylogenetic Analysis on Current HPC Architectures. *Scientific Programming*, 2,3(16):255–270, 2008.

[147] M. Ott, J. Zola, A. Stamatakis, and S. Aluru. Large-scale maximum likelihood-based phylogenetic analysis on the IBM BlueGene/L. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 4. ACM, 2007.

[148] C.-Y. Ou, C. A. Ciesielski, G. Myers, C. I. Bandea, C.-C. Luo, B. T. M. Korber, J. I. Mullins, G. Schochetman, R. L. Berkelman, A. N. Economou, J. J. Witte, L. J. Furman, G. A. Satten, K. A. Maclnnes, J. W. Curran, H. W. Jaffe, L. I. Group, and E. I. Group. Molecular Epidemiology of HIV Transmission in a Dental Practice. *Science*, 256(5060):1165–1171, 1992.

[149] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[150] S. Pääbo. The mosaic that is our genome. *Nature*, 421(6921):409–412, 2003.

[151] D. W. Page and L. R. Peterson. Reprogrammable PLA. United States Patents, Patent number 4,508,977, 1985.

[152] P. Pamilo and M. Nei. Relationships between gene trees and species trees. *Molecular biology and evolution*, 5(5):568–583, 1988.

[153] K. Papadimitriou, A. Anyfantis, and A. Dollas. An effective framework to evaluate dynamic partial reconfiguration in FPGA systems. *Instrumentation and Measurement, IEEE Transactions on*, 59(6):1642–1651, 2010.

[154] P. Pavlidis, J. D. Jensen, and W. Stephan. Searching for footprints of positive selection in whole-genome SNP data from nonequilibrium populations. *Genetics*, 185(3):907–922, 2010.

[155] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa. Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 9–17. IEEE, 2009.

[156] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. Numerical recipes in C. *The art of scientific computing (Cambridge: University Press*, 3(2), 1992.

[157] A. Rambaut and N. C. Grassly. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Comput Appl Biosci*, 13(3):235–238, 1997.

[158] B. Rannala and Z. Yang. Probability distribution of molecular evolutionary trees: a new method of phylogenetic inference. *Journal of molecular evolution*, 43(3):304–311, 1996.

[159] B. Redelings and M. Suchard. Joint Bayesian estimation of alignment and phylogeny. *Systematic Biology*, 54(3):401–418, 2005.

[160] J. Ripplinger and J. Sullivan. Does Choice in Model Selection Affect Maximum Likelihood Analysis? *Systematic Biology*, 57(1):76–85, 2008.

[161] F. Rodríguez, J. L. Oliver, A. Marín, and J. R. Medina. The general stochastic model of nucleotide substitution. *Theoretical Biology*, 142(4):485–501, 1990.

[162] J. Rogers. On the consistency of maximum likelihood estimation of phylogenetic trees from nucleotide sequences. *Systematic biology*, 46(2):354–357, 1997.

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

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

[165] F. Ronquist and J. P. Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics*, 19(12):1572–1574, 2003.

[166] M. Rosenberg and S. Kumar. Traditional phylogenetic reconstruction methods reconstruct shallow and deep evolutionary relationships equally well. *Molecular Biology and Evolution*, 18(9):1823–1827, 2001.

[167] V. Sachdeva, M. Kistler, E. Speight, and T. Tzeng. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. *Parallel Computing*, 34(11):616–626, 2008.

[168] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1984.

[169] N. Saitou and M. Nei. The number of nucleotides required to determine the branching order of three species, with special reference to the human-chimpanzee-gorilla divergence. *Journal of molecular evolution*, 24(1):189–204, 1986.

[170] M. Sanderson, M. McMahon, and M. Steel. Terraces in phylogenetic tree space. *Science*, 333(6041):448, 2011.

[171] M. J. Sanderson and M. J. Donoghue. Shifts in diversification rate with the origin of angiosperms. *Science*, 264(5165):1590–1593, 1994.

[172] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, pages 35–42, 1975.

[173] S. Sinha, J. Frahm, M. Pollefeys, and Y. Genc. GPU-based video feature tracking and matching. In *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, volume 278. Citeseer, 2006.

[174] B. Smith. boa: an R package for MCMC output convergence assessment and posterior inference. *Journal of Statistical Software*, 21(11):1–37, 2007.

[175] T. Smith and M. Waterman. Identification of common molecular subsequences. *Molecular Biology*, 147:195–197, 1981.

[176] R. R. Sokal and P. H. A. Sneath. *Principals of Numerical Taxonomy*. Freeman, San Francisco, 1963.

[177] A. Stamatakis. *Parsimonator*. http://www.exelixis-lab.org/software.html.

[178] A. Stamatakis. Phylogenetic models of rate heterogeneity: A high performance computing perspective. In *Proceedings of the 20th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS2006)*, 2006.

[179] A. Stamatakis. *TreeCounter*. http://www.exelixis-lab.org/software.html.

[180] A. Stamatakis. *Distributed and parallel algorithms and systems for inference of huge phylogenetic trees based on the maximum likelihood method*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2004.

[181] A. Stamatakis. Phylogenetic models of rate heterogeneity: A high performance computing perspective. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006.

[182] A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.

[183] A. Stamatakis. *Bioinformatics: High Performance Parallel Computer Architectures*, chapter Orchestrating the Phylogenetic Likelihood Function on Emerging Parallel Architectures, pages 85–115. CRC Press, Taylor & Francis, 2010.

[184] A. Stamatakis, A. Aberer, C. Goll, S. Smith, S. Berger, and F. Izquierdo-Carrasco. RAxML-Light: A Tool for computing TeraByte Phylogenies. *Bioinformatics*, 2012.

[185] A. Stamatakis and N. Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):i132–i139, 2010.

[186] A. Stamatakis, P. Hoover, and J. Rougemont. A Fast Bootstrapping Algorithm for the RAxML Web-Servers. *Systematic Biology*, 57(5):758–771, 2008.

[187] A. Stamatakis, Z. Komornik, and S. Berger. Evolutionary placement of short sequence reads on multi-core architectures. In *Proceedings of AICCSA-10, at 8th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-10), Hammamet, Tunisia*, 2010.

[188] A. Stamatakis, T. Ludwig, and H. Meier. RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. *Bioinformatics*, 21(4):456–463, 2005.

[189] A. Stamatakis and M. Ott. Efficient Computation of the Phylogenetic Likelihood Function on Multi-Gene Alignments and Multi-Core Architectures. *Philosophical Transactions of the Royal Society B*, 1512(363):3977–3984, 2008.

[190] A. Stamatakis and M. Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: a performance study. *Pattern Recognition in Bioinformatics*, pages 424–435, 2008.

[191] A. Stamatakis, M. Ott, and T. Ludwig. RAxML-OMP: An efficient program for phylogenetic inference on SMPs. *Parallel Computing Technologies*, pages 288–302, 2005.

[192] M. Stark, S. Berger, A. Stamatakis, and C. von Mering. MLTreeMap-accurate Maximum Likelihood placement of environmental DNA sequences into taxonomic and functional reference phylogenies. *BMC genomics*, 11(1):461, 2010.

[193] M. Suchard and B. Redelings. BAli-Phy: simultaneous Bayesian inference of alignment and phylogeny. *Bioinformatics*, 22(16):2047–2048, 2006.

[194] D. L. Swofford. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4*. Sinauer Associates, Sunderland, Massachusetts, 2003.

[195] D. L. Swofford, G. J. Olsen, P. J. Wadell, and D. M. Hillis. Phylogenetic inference. In D. M. Hillis, C. Moritz, and B. K. Mable, editors, *Molecular Systematics*, pages 407–514. Sinauer Associates, Sunderland, MA, 1996.

[196] Y. Tateno, M. Nei, and F. Tajima. Accuracy of estimated phylogenetic trees from molecular data. *Journal of Molecular Evolution*, 18(6):387–404, 1982.

[197] J. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[198] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.

[199] X. Tian and K. Benkrid. High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):26, 2010.

[200] A. Tumeo, M. Branca, L. Camerini, M. Ceriani, G. Palermo, F. Ferrandi, D. Sciuto, and M. Monchiero. A dual-priority real-time multiprocessor system on FPGA for automotive applications. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1039–1044. ACM, 2008.

[201] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Computational Biology*, 1(4):337–348, 1994.

[202] L. Weiguo, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1270–1281, 2007.

[203] D. G. Wheeler, W. C and J. D. Laet. POY, version 3.0.11. American Museum of Natural History, 2003.

[204] S. Whelan and N. Goldman. A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Evolution*, 18(5):691–699, 2001.

[205] T. Williams and B. Moret. An investigation of phylogenetic likelihood methods. In *Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium on*, pages 79–86. IEEE, 2003.

[206] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer applications in the biosciences: CABIOS*, 13(2):145–150, 1997.

[207] Xilinx Inc. *Virtex-6 Family Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[208] Xilinx Inc. *Virtex-6 FPGA Configurable Logic Block*. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.

[209] Z. Yang. Maximum likelihood estimation of phylogeny from DNA sequences when substitution rates differ over sites. *Molecular Biology and Evolution*, 10:1396–1401, 1993.

[210] Z. Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. *Molecular Evolution*, 39(3):306–314, 1994.

[211] Z. Yang. Among-site rate variation and its impact on phylogenetic analyses. *Trends in Ecology & Evolution*, 11(9):367–372, 1996.

[212] Z. Yang. Maximum Likelihood Estimation on Large Phylogenies and Analysis of Adaptive Evolution in Human Influenza Virus A. *Molecular Evolution*, 51:51–423, 2000.

[213] S. Yokoyama, W. Starmer, and R. Yokoyama. Paralogous origin of the red-and green-sensitive visual pigment genes in vertebrates. *Molecular biology and evolution*, 10(3):527–538, 1993.

[214] C. Yu, K. Kwong, K. Lee, and P. Leong. A Smith-Waterman systolic cell. *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pages 291–300, 2005.

[215] L. Zhou and W. Han. A brief implementation analysis of SHA-1 on FPGAs, GPUs and Cell processors. In *Engineering Computation, 2009. ICEC'09. International Conference on*, pages 101–104. IEEE, 2009.

[216] S. Zierke and J. Bakos. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC Bioinformatics*, 11(1):184, 2010.