Inference of Large Phylogenetic Trees on Parallel Architectures

Michael Ott

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur

Inference of Large Phylogenetic Trees on Parallel Architectures

Michael Ott

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. H. M. Gerndt
Prüfer der Dissertation:	1. Univ.-Prof. Dr. A. Bode
	2. TUM Junior Fellow Dr. A. Stamatakis

Die Dissertation wurde am 15.07.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.10.2010 angenommen.

Abstract

Due to high computational demands, the inference of large phylogenetic trees from molecular sequence data requires the use of HPC systems in order to obtain the necessary computational power and memory. The continuous explosive accumulation of molecular data, which is driven by the development of cost-effective sequencing techniques, amplifies this requirement additionally. Furthermore, a continuously increasing degree of parallelism is necessary in order to exploit the performance of emerging multi-core processors efficiently.

This dissertation describes scalable parallelization schemes for the inference of large phylogenetic trees as well as tangible implementations of those which also eliminate memory requirements as a limiting factor for phylogenetic analyses. Additionally, it pinpoints the properties of current multi-core shared and distributed memory architectures and describes novel approaches for their efficient exploitation.

Acknowledgments

Many people have contributed to the success of this work. First of all, I would like to thank Prof. Dr. Arndt Bode for providing such a supportive working environment at the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) and for the freedom he granted me for my research over the last years. I am particularly grateful to Dr. Alexandros Stamatakis who has been accompanying and supporting me since my undergraduate studies and spent a great effort on supervising my work. Furthermore, I am very thankful to the numerous helpful colleagues at the LRR – too many to be named individually – who have offered their advice when it was needed and supported me in many different ways.

My position at the LRR was funded by KONWIHR, the Bavarian Competence Network for Technical and Scientific High Performance Computing. KONWIHR is an initiative by the Bavarian State Ministry of Sciences, Research and the Arts for supporting research in computationally challenging scientific domains that require the use of High Performance Computing in order to obtain new knowledge and insights.

Contents

Chapter 1

Introduction

This chapter provides the motivation for developing methods and tools for the inference of large phylogenetic trees, summarizes the scientific contribution of the work, and outlines the structure of this thesis.

1.1 Motivation

The evolutionary history of all life forms on earth and their relation to each other – often referred to as *The Tree of Life* – has preoccupied mankind for centuries. While a comprehensive tree of life mainly serves human curiosity, phylogenetic trees at a lower scale that only comprise a smaller group of species have practical applications in many scientific fields. Besides their importance in systematic studies, phylogenetic trees allow for, predicting the evolution of infectious diseases [22] as well as the functions of uncharacterized genes [33] and can also be utilized for studying the dynamics of microbial communities [35]. Additionally, they play an important role in the discovery of new drugs [16] and vaccines [51, 64] and are even used in modern forensics [114].

As the number of potential applications of phylogenetic trees is steadily growing, there is an increasing demand for tools that allow for inferring such trees from molecular data like DNA sequences efficiently. This demand is amplified by the continuously growing amount of molecular data: recent advances in cost-effective high-throughput sequencing techniques such as, e.g., pyrosequencing [122] have generated an unprecedented molecular data flood. Public databases like GenBank [9] grow exponentially and with steadily decreasing costs for whole-genome sequencing – the 1,000$ human genome is within reach – this development is about to continue.

Unfortunately, the computational power of modern microprocessors does not keep pace with this data flood. Although Moore's law still holds and suggests that the performance of microprocessors doubles approximately every two years, there is an increasing gap – which we call the *bio-gap* – between the growth rates of molecular sequence data and microprocessor performance (see Figure 1.1). Additionally, over the last couple of years this performance increase was mainly achieved by the introduction of multi-core CPUs, i.e., by increasing the number

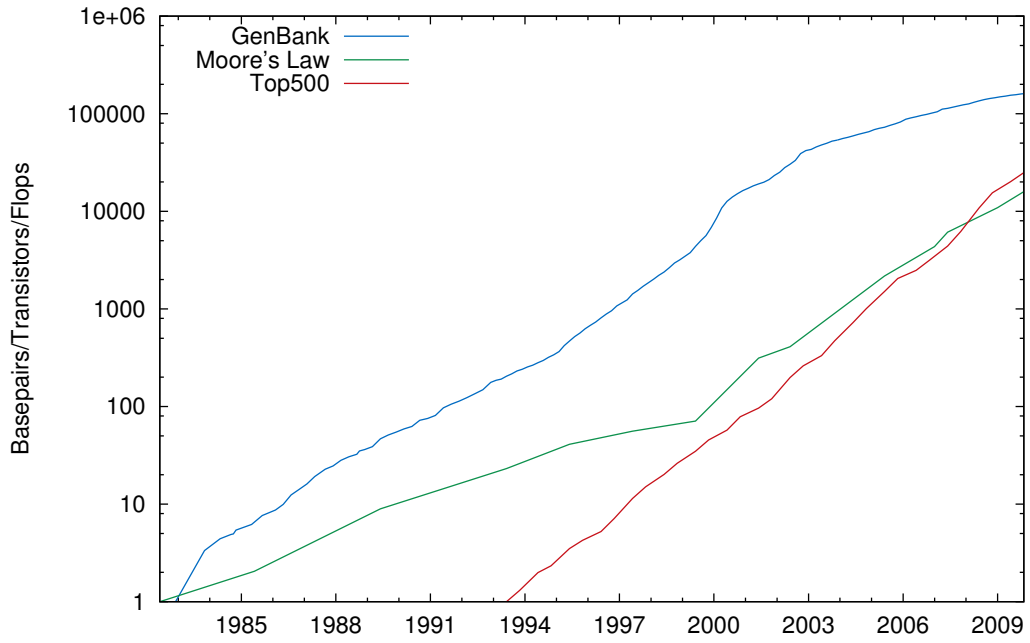

Figure 1.1: Growth of molecular data available in GenBank, of microprocessor performance according to Moore's Law, and of aggregate performance of the Top500 supercomputer list.

of cores per CPU while the actual performance per core remained constant. Consequently, sequential applications experienced almost no performance gain in recent years but only parallel applications that allow for utilizing multiple cores simultaneously were able to benefit from the latest developments in computer technology. Hence, in order to exploit the computational performance of modern multi-core processors for phylogenetic analyses, appropriate parallelization schemes need to be devised that allow for distributing the workload over multiple processor cores.

However, even a highly efficient parallelization approach for multi-core processors will not close the bio-gap but will only alleviate its impact. Closing the gap between computational power and data growth will require computer systems that experience performance increases at a rate that is significantly higher than the growth rate of molecular databases. The only class of systems that meets this criterion are high performance computing (HPC) systems. As can be seen from Figure 1.1, the aggregate performance of the systems in the Top500 supercomputer list [3] grows significantly faster than the performance of individual microprocessors and even outperforms the growth rate of GenBank. The only viable solution for closing the bio-gap therefore is to utilize HPC systems that provide the necessary computational power for dealing with the molecular data flood. Yet, the same trend that can been witnessed for microprocessors, also applies to supercomputers: exploiting the performance of those systems requires a continuously increasing level of parallelism of the utilized applications. As of June 2010 there is not a single system in the Top500 list with less than 1024 processor cores and the number one

system even comprises 224,162 cores. Utilizing such HPC resources for large-scale phylogenetic analyses therefore requires parallel applications that are able to scale up to thousands of cores.

Furthermore, the huge amount of molecular data that is to be used for phylogenetic analyses poses challenges not only in terms of computational performance but also in terms of memory requirements. Given that the memory footprints of large-scale analyses can easily exceed 100 GB and that due to high costs, computers are typically equipped with less than 128 GB of main memory, memory requirements are effectively a limiting factor for phylogenetic analyses. This also applies to supercomputers, although they have a significantly larger amount of aggregate main memory. However, modern supercomputers are typically designed as distributed memory architectures that comprise hundreds or thousands of compute nodes, each of which is equipped with a rather low amount of memory (typically less than 32 GB). Any parallelization approach that aims at facilitating such large-scale analyses therefore needs to distribute not only the computational workload over thousands of cores but also the data structures that are required for the computations over multiple distributed memory nodes.

Besides the application-specific challenges, the parallel computer architectures that have emerged over recent years pose new challenges of their own for the development of highly efficient applications: the multi-core revolution led to parallel microprocessors and computer systems with heterogeneous properties that need to be accounted for in order to avoid performance bottlenecks. The massive parallelism that is required for exploiting the performance of state-of-the-art HPC systems requires new approaches not only in terms of highly parallel applications but also in terms of scalable development and debugging tools.

1.2 Scientific Contribution

This dissertation is an interdisciplinary work in Bioinformatics and Computer Science and as such comprises scientific contributions to both domains. These contributions can generally be divided into three categories:

Firstly, the characteristics of emerging multi-core architectures have been thoroughly analyzed and potential pitfalls and bottlenecks that could impact application performance on such systems have been described. Additionally, the `autopin` framework has been developed which facilitates automated and performance-aware thread placement on multi-core systems and hence maximizes the parallel efficiency on such systems without user interaction.

Secondly, a scalable approach for the automatic detection of performance bottlenecks in large-scale MPI (Message Passing Interface) applications that is solely based on summary information has been developed and integrated into the `Periscope` tool for automatic performance analysis.

Thirdly, a generic parallelization approach for Maximum Likelihood based tools for phylogenetic tree inference has been presented and implemented in `RAxML`. It allows for exploiting loop-level parallelism in the phylogenetic likelihood function on shared memory as well as distributed memory architectures and overcomes memory requirements as a limiting factor for phylogenetic analyses.

Furthermore, the MPI-based parallelization of `RAxML` has been used to conduct the most computationally intensive phylogenetic study to date [68]: over 2.25 million CPU hours have been utilized for analyzing several datasets of 94 bilaterian animals, the largest one comprising 270,580 basepairs in 1,487 genes (see respective press releases by Technische Universität München on October 29, 2009 and Brown University on September 23, 2009). The aim of the study was to assess the root of bilaterian animals and was conducted in co-operation with 15 researchers from the United States, France, Germany, Sweden, Spain, and the United Kingdom.

Moreover, the MPI parallelization has recently been included as `125.RAxML` in the SPEC MPI2007 v2.0 benchmark suite [102, 110] which is used for assessing the performance of high performance computers. Technische Universität München and Leibniz Computing Centre jointly published a respective press release on April 14, 2010 on this occasion, that has also been covered by the popular press.

The scientific results of this thesis have been incrementally published in eight peer-reviewed conference papers [141, 152, 112, 113, 109, 139, 140, 110] and five journal articles [111, 86, 138, 52, 68]. The papers are available for download in PDF format at `http://www.lrr.in.tum.de/~ottmi`.

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 data. Chapter 3 gives an overview of the properties of (multi-core) shared memory systems and describes approaches for exploiting their performance efficiently. The Characteristics of distributed memory systems as well as the prevailing program paradigm (MPI) and concepts for performance optimizations on those systems are discussed in Chapter 4. Potential sources of parallelism in phylogenetic analyses and tangible parallel implementations are described in Chapter 5. The performance of these implementations is evaluated in Chapter 6 and Chapter 7 provides a conclusion of the thesis and gives an outlook on future work.

Chapter 2

Phylogenetic Tree Inference

This chapter introduces the basic concepts of phylogenetic tree inference and the associated optimization problem. Furthermore, it describes the most commonly used methods and how they model the evolutionary process. Finally, it provides an overview over state-of-the-art tools for inferring phylogenetic trees.

2.1 Introduction

Phylogenetic trees represent the evolutionary history of a set of organisms. In traditional phylogenetics, which more or less date back to the 19th century and Charles Darwin's seminal book *The Origin of Species* [28], morphological data was used to derive the evolutionary relationship between individual organisms. By its nature, such data is imprecise and for many organisms it is hard if not impossible to acquire: while it might be possible to classify some species by certain distinguishing characteristics like the presence of wings, shape of bones, size of certain organs, etc., such characterizing properties may be hard to define for very small organisms like bacteria or viruses. Moreover, morphological properties may not allow for comparing such small organisms with complex organisms like mammals or for comparing very morphologically distant organisms in general. Additionally, the assumption that similar body parts imply a relationship between two species may be misleading, as they could also have developed from independent lineages (so-called homoplasy). For example, bats and sparrows both have wings, but they descended from very different lineages: sparrows belong to the class of birds while bats belong to the class of mammals.

In contrast, molecular phylogenetics utilizes nucleotide sequences which can encode genes or amino acid sequences which can encode proteins to classify organisms. Such sequences can be used as input data for computational phylogenetic algorithms that allow for deriving a phylogenetic tree of the species represented by these sequences. In general, these sequences are composed of a series of characters drawn from a limited alphabet. There are 4 possible characters in the case of nucleotide sequences (A, C, G, T for DNA data and A, C, G, U for RNA data) and 20 possible characters ($A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y$) in the

case of amino acid sequences.[1]

Though some alignment-free methods exist (see [75] for an overview), most methods for molecular tree inference require the input data to be provided as multiple sequences alignment, which is a matrix whose lines consist of the sequences of the individual organisms. The sequences have to be aligned to each other such that all sites (i.e., the columns of the matrix) are homologous, which means that all characters are related by a common evolutionary history. Finding an optimal multiple sequences alignment is a NP-hard problem [151]. Multiple algorithms and heuristics have been proposed for this task, ranging from dynamic programming approaches [92], progressive techniques [105, 147], and iterative methods [18, 57] to hidden Markov models [74] and genetic algorithms [104]. However, all of these approaches are beyond the scope of this thesis and will therefore not be discussed. For the rest of this chapter we assume that the multiple sequences alignments we use as input for the phylogenetic tree inference are given. Keep in mind that the quality of the obtained tree topology highly depends on the quality of the input alignment.

In the context of computational phylogenetics, a phylogenetic tree is usually an unrooted, strictly bifurcating tree: every node has either none (terminal nodes) or exactly two child nodes (internal nodes). The terminal nodes (also called tips or leaves) of the tree represent the organisms for which we have sequences in the input data set, i.e., the organisms in whose phylogeny we are actually interested in. The internal nodes represent hypothetical extinct ancestors. Evolutionary events such as mutations of sites are modeled to occur along the branches of the tree, between ancestor and descendant. Depending on the method that is used for the inference of the phylogenetic tree, the branches of the tree may be weighted. In this case they could provide information, e.g., about the time that was needed for the sequence at the parent node to develop into the sequence at the child node.

Unrooted trees only describe the relatedness of the species represented by the molecular sequences at the terminal nodes of the tree topology. As opposed to rooted trees, they do not allow for any assumption about a single common ancestor which is contrary to the intuitive expectations one would have about a phylogenetic tree, especially a *Tree of Life*. Though it is easily possible to convert a rooted tree into an unrooted tree simply by removing the root, the contrary direction is not as straightforward. As every branch could be designated the root of the tree, additional knowledge about the ancestral relationships is needed to pick a reasonable one. It is common practice to add an additional so-called outgroup to the alignment and use this outgroup for rooting the tree after the inference has been completed. The outgroup species must not be closely related to any other species of the tree in order not to disturb the tree inference of the species one is actually interested in. However, using a too distant outgroup may also bias the phylogenetic analysis. For a detailed discussion on outgroup choice see [59].

[1] The respective alphabet may be larger if ambiguous character encoding is allowed.

2.2 The Optimization Problem

Reconstructing phylogenetic trees from molecular data is a combinatorial optimization problem: given a certain number of feasible solutions and a scoring function that evaluates the quality of a solution according to some optimality criterion, we are searching for the solution that achieves the highest score.

In the context of phylogenetic tree inference, any tree topology that contains the species of interest is a feasible solution. That is, any binary unrooted tree that has all species assigned to its leaves is a potential candidate for the optimal phylogenetic tree. Thus, for a given sequence alignment containing n species, the feasible solutions are all binary unrooted trees with n leaves.

The scoring function evaluates a given tree topology and reduces its complexity to a single numerical value that is a measure for the fit of the tree to the data. Every scoring function relies on a model of evolution that describes the process of a molecular sequence evolving over time (see Section 2.3). The numerical value returned by the scoring function allows for comparing two tree topologies and deciding which topology fits the given molecular sequences better under the assumed model of evolution.

Besides the actual evolutionary model utilized, scoring functions can additionally be divided into two classes: distance methods and character-based methods. Distance methods initially compute pairwise distances of the molecular sequences and perform all following operations on these numerical values only. Thus, they loose information about the molecular sequences which might affect the trustworthiness of the obtained result. In contrast, character-based methods utilize the molecular sequences as a whole through the entire process. In general, distance methods are considered to be faster but less accurate than character-based methods. Actually, only two character-based methods are regularly used for phylogenetic tree inference: Maximum Parsimony (MP, see Section 2.5) and Maximum Likelihood (ML, see Section 2.6). The best known distance methods are described more closely in Section 2.4.

There have been extensive discussions over the last 25 years about the superiority of one method over the other. However, it is now widely accepted that Maximum Likelihood-based approaches obtain more accurate results [72, 124, 164]. Yet, this accuracy comes at a high price: Maximum Likelihood (ML) is by far the most expensive method regarding computational resources – memory as well as computations. Consequently, the computationally inexpensive distance methods or the less elaborate Maximum Parsimony approach are still used under some circumstances, for example for analyzing huge datasets whose computational requirements are prohibitive for more elaborate methods.

Regardless of the actual scoring function used, the algorithmic challenge in phylogenetic tree inference is due to the huge solution set of the optimization problem. The scoring function only gives a numerical measure for the fit of the tree to the input data. It does not provide any hint whether there is a better tree or how to optimize a given tree in order to improve the score. Hence, the naive approach for finding the optimal tree would be to create all feasible topologies and evaluate their quality by means of the scoring function. Unfortunately, the number of possible tree topologies grows factorially with the number of species. For n species the number

of possible binary unrooted trees $T(n)$ is [31]:

$$T(n) = \prod_{i=3}^{n}(2i-5) \tag{2.1}$$

A few exemplary figures for this formula are given in Table 2.1. Already for 50 species T becomes unimaginably large. Thus, enumeration and evaluation of all potential trees is not an option, regardless of the actual scoring function used. In fact, it has been shown for Maximum Likelihood [25] as well as for Maximum Parsimony [29] that finding the optimal tree is a NP-hard problem. Hence, the only feasible approach is to use heuristics that approximate the optimal solution. However, heuristics are – by their nature – not guaranteed to actually find it.

# Species	# Possible Tree Topologies
3	1
4	3
5	15
6	105
7	945
10	$2,027,025$
15	$7,905,853,580,625$
20	$2.21 \cdot 10^{20}$
50	$2.84 \cdot 10^{76}$

Table 2.1: Number of possible tree topologies for 3–50 species

As for most optimization problems, numerous heuristics have been proposed and implemented in different programs for phylogenetic inference. Section 2.9 gives an overview over several state-of-the-art programs and the heuristics they implement. Distance methods often use greedy algorithms that build a tree by consecutively adding new nodes to the topology according to the scoring function used. Discrete methods frequently implement hill-climbing algorithms: initially they create a tree topology and reorganize it step by step in order to improve its quality or more precisely, its score under the given optimality criterion. The typical standard topological rearrangement moves which are used for this task are *Nearest Neighbor Interchange (NNI)*, *Subtree Pruning and Re-grafting (SPR)*, and *Tree Bisection and Reconnection (TBR)*. In general all of these moves result in removing a sub-tree and inserting it at another position in the tree such that the score of the tree topology improves. A detailed description of these rearrangement moves is given by Swofford et. al in [145].

2.3 Models of Sequence Evolution

Every phylogenetic analysis relies on a model of sequence evolution, i.e., a model that describes the process of one sequence evolving into another. Maximum Parsimony-based approaches assume a model of minimum evolution that mainly considers the number of differing sequence

characters when determining the relatedness of two species. This specific model will be described in Section 2.5 along with the Maximum Parsimony approach itself. In contrast, most distance methods as well as Maximum Likelihood-based approaches utilize explicit probabilistic evolutionary models. These models usually assume individual sequence sites to be evolving independently of each other. The change at any particular site from one character state to another state is commonly modeled using continuous-time Markov chains with the potential sequence states being the states of the chain (e.g. `A,C,G,T` for DNA sequences). The main feature of a Markov chain is the so-called Markovian property: it has no memory. Thus, the probability of a chain changing its state solely depends on its current state. Any state that the chain might have had previously will not affect any future state change. Additionally the state changes of a sequence site are assumed to be time-reversible, which means that the evolutionary process is identical regardless in which direction of time it is followed: if a site has state s at present, the probability that it has emerged from state r in the past is identical to the probability that it will emerge into state r in the future. Note that the reversibility is assumed because of mathematical reasons and not because of biological reasons. Though non-reversible models have also been proposed (see, e.g., [13]), this thesis will concentrate on reversible models as most programs for phylogenetic inference require this property for an efficient implementation.

Appropriate models have been developed for various molecular sequence types such as nucleotide sequences (see next Section) as well as amino acid [30, 83, 154] or codon sequences [55, 103]. In general, phylogenetic analyses are not restricted to molecular sequences. Any sequence type that sufficiently characterizes a species and for which a biologically meaningful model of sequence evolution is available is suitable for phylogenetic analyses. As long as the above listed properties are met (independently evolving sites, Markovian property, time-reversibility), new models can easily be added to existing applications for phylogenetic analyses. For example, discrete morphological data can also be used for computational phylogenetic tree inference [90].

2.3.1 The Substitution Rate Matrix

The process of molecular evolution is modeled by a continuous-time Markov process. Such a process is defined by the transition rates q_{ij} that describe the instantaneous rate of change from chain state i to j: let $X(t)$ be the state of the chain at time t. If the chain is in state i at time t, the probability that it has made the transition to some other state $j \neq i$ after time Δt is given by

$$Pr\{X(t + \Delta t) = j | X(t) = i\} = q_{ij}\Delta t. \tag{2.2}$$

The transition rates q_{ij} are specified by the *transition rate matrix* Q which is often also called *substitution rate matrix* or *instantaneous rate matrix* in the context of models for molecular evolution. The matrix has dimension $n \times n$ where n is the number of possible states of the data. For simplicity, we will concentrate on models for DNA data in the following, i.e., $n = 4$ and the possible character states are `A,C,G,T`. However, analogous models are available for amino acid sequences ($n = 20$) as well as codon sequences ($n = 61$). The principles described in this

chapter apply to them accordingly. The j-th entry of the i-th row of the matrix corresponds to the transition rate q_{ij}.

The most general form of Q for DNA data is given by equation (2.3) (rows and columns are ordered by the alphabetical order of the bases `A,C,G,T`):

$$Q = \begin{pmatrix} -\mu(a\pi_C + b\pi_G + c\pi_T) & \mu a\pi_C & \mu b\pi_G & \mu c\pi_T \\ \mu g\pi_A & -\mu(g\pi_A + d\pi_G + e\pi_T) & \mu d\pi_G & \mu e\pi_T \\ \mu h\pi_A & \mu j\pi_C & -\mu(h\pi_A + j\pi_C + f\pi_T) & \mu f\pi_T \\ \mu i\pi_A & \mu k\pi_C & \mu l\pi_G & -\mu(i\pi_A + k\pi_C + l\pi_G) \end{pmatrix} \tag{2.3}$$

The parameter μ represents the mean instantaneous substitution rate from one nucleotide to another. The relative rate parameters $a, b, ..., l$ correspond to each of the possible 12 transformation types between distinct bases. The product of the mean instantaneous substitution rate and a relative rate parameter constitutes a *rate parameter*. However, μ is usually set to 1 and $a, b, ..., l$ are scaled such that the average substitution rate is 1. It is assumed that the nucleotide substitutions are at equilibrium, i.e., the proportions of the individual bases stay constant over time. Thus, the overall base composition of the input alignment, i.e., the occurrence frequency of the individual bases `A,C,G,T` – often called base frequencies – as represented by the parameters $\pi_A, \pi_C, \pi_G, \pi_T$ also serves as the stationary distribution of the Markov process $\pi = (\pi_A, \pi_C, \pi_G, \pi_T)$ and constitutes the prior probabilities. The diagonal elements of Q are chosen such that every row sums up to zero, i.e., $q_{ii} = -\sum_{i \neq j} q_{ij}$. Thus $-q_{ii}$ describes the rate at which the Markov chain leaves state i, that is, the substitution rate of nucleotide i.

The proportion of time the Markov chain spends in state i is described by π_i, whereas q_{ij} is the rate at which the chain's state changes from i to j. Hence, $\pi_i q_{ij}$ describes the amount of flow from state i to j and $\pi_j q_{ji}$ the flow in the opposite direction. As discussed above, the Markov chain used to model the molecular substitution process usually needs to be time-reversible. According to the *detailed-balance* condition, this requires the following equation to hold:

$$\pi_i q_{ij} = \pi_j q_{ji}, \text{ for all } i \neq j. \tag{2.4}$$

However, the matrix of equation (2.3) does not meet this requirement as the relative rate parameters are not symmetric, e.g.,

$$\pi_A q_{AC} = \pi_A \mu a \pi_C \neq \pi_C \mu g \pi_A = \pi_C q_{CA}. \tag{2.5}$$

Consequently, we need to modify Q accordingly by setting $g = a, h = b, i = c, j = d, k = e, l = f$, i.e., reducing the number of rate parameters and hence enforcing symmetry:

$$Q = \begin{pmatrix} -(a\pi_C + b\pi_G + c\pi_T) & a\pi_C & b\pi_G & c\pi_T \\ a\pi_A & -(g\pi_A + d\pi_G + e\pi_T) & d\pi_G & e\pi_T \\ b\pi_A & d\pi_C & -(h\pi_A + j\pi_C + f\pi_T) & f\pi_T \\ c\pi_A & e\pi_C & f\pi_G & -(c\pi_A + e\pi_C + f\pi_G) \end{pmatrix} \tag{2.6}$$

Note that, additionally μ has been set to 1 as described above for better readability. The evolutionary model described by this simplified matrix is known as the *General Time-Reversible (GTR) model* [88]. It allows for four distinct base frequencies and 6 rate parameters. Other models have been proposed which utilize an even more simplified matrix to reduce computational

demands. This is achieved by further reducing the number of rate parameters and/or the number of base frequencies. The best known models are *JC69* by Jukes and Cantor [84] (1 rate parameter, equal base frequencies), *F81* by Felsenstein [42] (1 rate parameter, 2 base frequencies), *K2P* by Kimura [85] (2 rates, equal base frequencies), and *HKY85* by Hasegawa, Kishino, and Yano [66] (2 rates, 4 base frequencies). However, these simple models have rarely been used for recent phylogenetic analyses [121] as GTR now is feasible thanks to efficient software implementations and powerful CPUs.

2.3.2 The Substitution Probability Matrix

As mentioned above, any entry q_{ij} of the instantaneous rate matrix Q specifies the probability that a given nucleotide i will change to a different nucleotide j in an infinitely small time interval Δt (eq. (2.2)). However, to describe the process of molecular evolution sufficiently, we need a *substitution probability matrix* over time $t > 0$: $P(t) = \{p_{ij}(t)\}$ with

$$p_{ij}(t) = Pr\{X(t) = j | X(0) = i\}. \tag{2.7}$$

$P(t)$ describes the probability that a nucleotide in state i will evolve to state j after time t and is the solution to the differential equation

$$\frac{dP(t)}{dt} = P(t)Q, \tag{2.8}$$

with the boundary condition $P(0) = I$, the identity matrix. This has the solution

$$P(t) = e^{Qt}. \tag{2.9}$$

The exponential can be evaluated by decomposing Q into its eigenvalues and eigenvectors. For the simpler models like JC69, K2P, or HKY85 there exist closed form expressions of P that allow for a direct analytical computation of the matrix elements. For GTR however, the decomposition of Q has to be solved numerically as no closed form expression exists. See [89] for a closer discussion of potential numerical decomposition methods.

2.3.3 Rate Heterogeneity

The GTR model described above as well as most other evolutionary models assume that all sites of the molecular sequences evolve at the same rate. Yet, this assumption over-simplifies the complex evolutionary process. For example, in case of protein coding genes, the third nucleotide position of codons usually evolve faster than the first and second position. Additionally, some sites may be resistant to change due to strong selective constraints: the mutation of a specific site may result in the loss of a unique property that the species needs to survive in its ecological niche. The mutated gene would therefore never be observed in the wild. Hence, accounting for rate heterogeneity is important when modeling the evolutionary process. In fact, several studies have shown that ignoring rate variation among sites can have a devastating impact on

the results of phylogenetic analyzes [15, 49, 159].

Rate heterogeneity can be incorporated into evolutionary models by modifying the substitution rates by an additional relative rate component r, i.e., by multiplying Q by r. Note that, only r varies from site to site, whereas Q is fixed. The respective substitution probability matrix $P(t, r)$ can then be calculated with

$$P(t, r) = e^{rQt} \tag{2.10}$$

In the simplest case, each site k is assigned an individual rate r_k. Typically, such an assignment would be based on some a priori classification of sites into functional categories and an assignment of relative rates to these categories. For example, in case of the above mentioned protein coding genes, the categorizations might be (first, second) and third codon position. However, finding suitable categorizations and manually estimating the relative rates of the categories can be difficult and may also bias the phylogenetic analysis. Hence, several approaches have been proposed to automate this process. G. Olsen developed a program for nucleotide sequences which performs a maximum likelihood estimate of the individual per site rates for a fixed tree topology (DNArates [107]). Additionally, there are several stochastic models available that account for rate heterogeneity. In these models, the rate r for any site is a random variable drawn from a – either discrete or continuous – statistical distribution.

2.3.3.1 Discrete-Rates Model

In case of the discrete distribution of rates it is assumed that sites fall into K classes of different rates. Each site falls into class k and thus has the rate parameter r_k with probability p_k, $k = 1, 2, ..., K$. The probabilities sum to 1 ($\sum p_k = 1$) and the average rate is $\sum p_k r_k = 1$.

A special case of a discrete-rates model is the *invariable-sites* model as implemented in HKY85, which assumes two classes of sites: variable sites and invariable sites. The rate of invariable sites is $r_0 = 0$ and their proportion is p_0. As the average rate needs to be 1, the rate of variable sites is $r_1 = (1 - p_0)^{-1}$.

2.3.3.2 Gamma-Rates Model

The best known continuous distribution for modeling rate variation among sites is the Γ distribution model proposed by Z. Yang [157]. Note that, the choice of the Γ distribution is arbitrary and any other continuous distribution could be used. However, the use of the Γ distribution has meanwhile been well studied and proven to be appropriate.

The Γ density function is

$$g(r; \alpha, \beta) = \frac{\beta^\alpha r^{\alpha-1} e^{-\beta r}}{\Gamma(\alpha)}. \tag{2.11}$$

Generally, the Γ distribution is parametrized by the shape parameter α and the scale parameter β. However, β is set to α in order to obtain a mean distribution rate of 1 and thus maintain Q's mean substitution rate of 1. The α parameter is inverse to the coefficient of variation of the substitution rate, i.e., a low α means high rate variation, a high α means low rate variation. For $\alpha \to \infty$ the distribution degenerates into a model of a single rate for all sites.

2.3.3.3 Discrete-Gamma Model

Yang also proposed an approximation for the Gamma-Rates Model, known as the discrete Γ model [158]. In this model, the continuous rate distribution is approximated with a discrete distribution: the sites are divided into K equally probable rate categories, i.e., $p_k = 1/K$. The rate r_k of each category is represented by the mean of the rates falling into that category.

Yang showed that approximating the Γ distribution by 4 discrete rates, achieves similar results for most real world datasets, but significantly reduces computational requirements. Consequently, the discrete Γ model is used instead of the continuous Γ model in most programs for phylogenetic inference.

2.4 Distance Methods

Distance methods do not use the actual molecular sequence alignment during the tree inference but calculate a symmetric $n \times n$ matrix from the input alignment in the beginning. The entries of this matrix are the pairwise-distances of the n sequences. The actual tree inference is then performed solely on the basis of this matrix. For computing the matrix a distance function $\delta(S_i, S_j)$ $i, j = 1, ..., n$ is required that provides a measure for the genetic distance of each pair of the n sequences in the input alignment. In the simplest case this function would only count the number of differing characters of the two sequences. More elaborate functions, however, utilize a sophisticated model of molecular evolution as described in the previous section. The most frequently used distance-based approaches are probably the LS (Least-Squares) method [23] and the UPGMA (Unweighted Pair Group Method with Arithmetic Mean) [130] and NJ (Neighbor-Joining) [126] heuristics.

2.4.1 Least-Squares

The Least-Squares method estimates the branch lengths of a tree topology by matching the distances described by them as closely as possible to the values of the pairwise distances matrix. This is achieved by minimizing the sum of squared differences between the given (by the distances matrix) and the predicted distances. The predicted distance between two sequences is calculated as the sum of the branch lengths along the path connecting both of them. The sum of all squared differences represents a measure for the fit of the tree to the given sequence data: the tree with the minimal sum is the optimal tree. The complexity of LS is $O(n^3)$.

2.4.2 UPGMA

UPGMA is a clustering algorithm that builds a rooted tree topology by stepwise addition. A molecular clock is assumed for the evolutionary process, which means that all species contained in the phylogenetic tree are supposed to evolve at the same rate. This assumption leads to the fact that trees obtained by UPGMA are ultrametric trees, that is, all end nodes (representing the species of interest) are equidistant from the root.

The algorithm works as follows: In the beginning, each node represents a cluster. At each step, the two clusters whose associated sequences have minimal distance according to the distance matrix are joined. Their entries are removed from the matrix and an entry for the new cluster is added. The distance of the new cluster to other clusters is computed as the mean distance of the sequences contained in each cluster. The algorithm terminates when all clusters have been joined into a single cluster. The complexity of UPGMA is $O(n^2)$.

2.4.3 Neighbor-Joining

Neighbor-Joining is also a clustering algorithm and is based on the minimum-evolution criterion. The tree that explains the sequence data with the minimal amount of change, i.e., the tree which minimizes the sum of all branch lengths (the total tree length), is the optimal tree.

The algorithm starts with a star-tree. At each step, two nodes are removed from the tree and reconnected via a common newly added internal node. The distance of both nodes to any other node of the tree (i.e., the sum of the branch lengths on the path connecting the nodes) stays constant. Yet, the total tree length is reduced as two rather long branches are replaced by three shorter branches. The nodes to be reorganized are selected such that the greatest reduction of the tree length is achieved. This procedure is repeated until the tree is fully resolved.

The complexity of the original NJ implementation is $O(n^3)$ which can be reduced to $O(n^2)$ by using a more sophisticated algorithm for selecting the nodes to be joined [162].

2.4.4 Objections Against Distance Methods

The main objection against distance methods is the fact that transforming the input alignment to a simple distance matrix at a very early stage of the tree reconstruction process evidently neglects information which might be valuable for inferring the relationships between the species being analyzed.

Furthermore, UPGMA as well as NJ are greedy algorithms. By their nature, such algorithms behave optimally only locally at each step of the iterative process. The final result, however, might not be optimal with regard to the underlying optimality criterion.

The molecular clock that is assumed by UPGMA probably oversimplifies the evolutionary process. While this assumption might be suitable for closely related species, the clock is often violated if the sequences are divergent.

The minimum-evolution criterion applied by NJ implies that mutations of molecular bases are always minimal, e.g., that a DNA mutation $A \rightarrow C$ always happens directly and not via a detour $A \rightarrow G \rightarrow C$. Potentially unobserved mutations (e.g. $A \rightarrow G \rightarrow A$) are also neglected.

In the strict sense, the Least-Squares method is not a method for inferring a tree topology for a given sequence alignment but for evaluating the fit of a given topology to the alignment. Consequently, it additionally requires a proposal algorithm that provides suitable tree topologies for evaluation. As pointed out in Section 2.2, the number of possible tree topologies grows factorially with the number of sequences. Though LS is a distance-based method, the main

benefit of these methods – their speed – is negligible as the computational complexity of a tree inference under LS will be imposed by the search algorithm.

2.5 Maximum Parsimony

Maximum Parsimony [45] favors the tree topology which explains the given data (the multiple sequences alignment) with the least amount of change, i.e., the lowest number of nucleotide or amino acid substitutions. In this sense, it is similar to the minimum-evolution criterion of NJ. However, MP computes the distance between two sequences on a per-column (per-site) basis and considers only so-called informative sites. Those are the columns of the sequence alignment that contain at least two different kinds of characters, each of which is represented in at least two of the sequences. The distance between two sequences is the number of differing characters at informative sites and is attributed as weight to the branch connecting the two sequences. For the inner nodes of the tree hypothetical sequences are calculated such that the distances between an inner node and its adjacent nodes are minimal. The Maximum Parsimony score of a tree can be calculated by summing up the weights of all branches. The tree with minimal score is the most parsimonious tree and thus the optimal tree under the Maximum Parsimony optimality criterion.

Since the Maximum Parsimony criterion is very similar to the minimum-evolution criterion, it also suffers from identical shortcomings (see Section 2.4). Additionally, the phenomenon of so-called *long branch attraction* [41] can be observed on MP-inferred phylogenies: sequences which are connected to the tree by very long branches, might be grouped together though they developed from very different lineages. Long branches indicate a high rate of change, i.e., the sequence at the terminal node of the branch differs from the hypothetical sequence at the internal node in many sites. Maximum Parsimony only accounts for the fact that *some* substitution took place at a specific site and not *which* substitution. Thus, it groups the two nodes with the long branches together solely because both highly differ from the other sequences. The fact that both of them also are highly different to each other is neglected.

Nevertheless, Maximum Parsimony is still frequently used for phylogenetic inference for several reasons [56]: Firstly, it is a character-based method and as such considered to be superior to distance methods at it uses all information that is contained in the input alignment for the tree reconstruction. Secondly, it is fast and therefore an alternative to Maximum Likelihood for large-scale datasets if computational resources are restricted. Thirdly, the phenomenon of *long branch attraction* is only an issue for small datasets. Fourthly, many biologists appreciate the fact that MP only makes few assumptions about the evolutionary process besides evolutionary change being rare.

2.6 Maximum Likelihood

In general, Maximum Likelihood is a parametric statistical method for fitting a mathematical model to some data. The principle of likelihood suggests that the explanation that makes the

observed outcome the most likely occurrence is the one to be preferred. Formally, given some data D and a hypothesis θ, the likelihood of that data is given by

$$L(D;\theta) = f(D|\theta) \tag{2.12}$$

which is the probability of obtaining D given θ. Though both terms are colloquially used synonymously, it is important to distinguish between probability and likelihood here. Informally, probability allows one to predict unknown outcome based on known parameters, whereas likelihood allows one to predict unknown parameters based on known outcome. In the context of phylogenetic inference, D is the (known) input alignment and the (unknown) hypothesis θ the combination of a phylogenetic tree and a model of molecular evolution along with its parameters. Thus, $L(D;\theta)$ describes how likely it is that the observed molecular sequences contained in the input dataset could have emerged from a specific phylogenetic tree under a specific model of evolution. More formally, θ comprises a tree topology τ, a parameter vector ν containing all branch lengths and a vector ϕ of evolutionary model parameters.

The absolute likelihood value has no specific meaning and does not allow for any conclusion whether the given phylogenetic tree actually is the true tree. It is only meaningful for comparing several trees to each other: the one with the highest likelihood value explains the given data the best. Thus, the two problems which need to be resolved are:

1. Which parameters ν and ϕ make the observed data D more likely for a given tree topology τ ?

2. Which tree topology τ_i of all possible tree topologies $\tau_{1..n}$ yields the highest likelihood ?

In order to find the optimal phylogenetic tree – under this specific optimality criterion – one would have to create and evaluate all possible tree topologies. As already pointed out in Section 2.2, the number of possible tree topologies grows factorially with the number of organisms in the input dataset. Evaluating all possible tree topologies, i.e., optimizing ν and ϕ and calculating the likelihood score, therefore is not an option for more than 10 organisms. Instead, elaborate heuristics are required that significantly reduce the search space but still approximate the optimal tree topology. Some approaches to this problem will be discussed in more detail in Section 2.9.

2.6.1 The Phylogenetic Likelihood Function

Calculating the actual likelihood score of a given tree topology is the essential step to infer trees under the Maximum Likelihood criterion. This computation is carried out according to Felsenstein's *pruning algorithm* [42]: at each node k, the conditional probability $L_{s_k}^{(k)}$ is calculated, which describes the probability of observing the data at the descendants of node k given the data at node k is s_k, with $s_k \in \{A, C, G, T\}$. As mentioned in Section 2.1, the species contained in the input sequence alignment along with their actual sequence data are stored at the tips of the tree. These are called *operational taxonomic units (OTU)*. The internal nodes

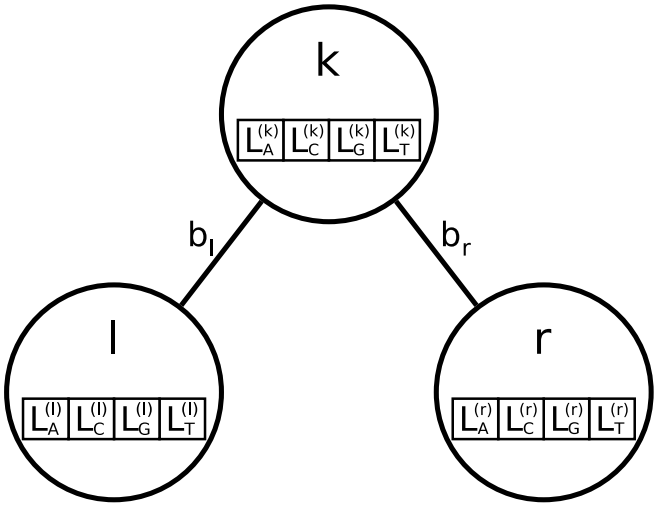

Figure 2.1: Computation of the conditional probabilities vector at an internal node k from the vectors of its child nodes l and r.

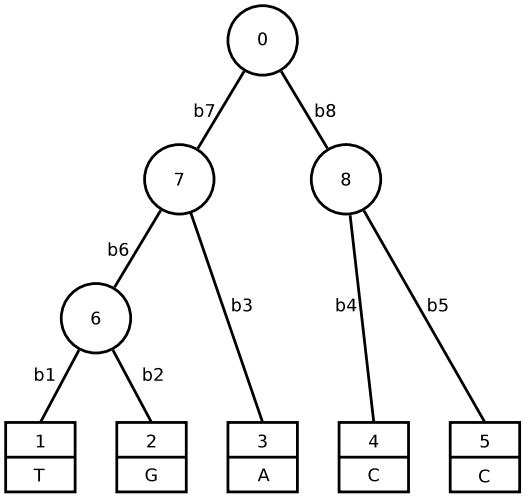

Figure 2.2: A rooted 5 taxon tree

represent common, probably extinct ancestors for which we have no sequence data and are hence called *hypothetical taxonomic units (HTU)*. The unavailability of sequence data for the internal nodes is the reason why we need to compute the conditional probabilities. Casually speaking, we consecutively assume that k has state A, C, G, T and compute the respective probabilities of the child nodes effectively having the nucleotide states as given by the input alignment. So we need to compute four conditional probabilities $L_{s_k}^{(k)}$ – one for each of the four feasible nucleotide states – which are combined to a vector of conditional probabilities $\vec{L}^{(k)} = [L_A^{(k)}, L_C^{(k)}, L_G^{(k)}, L_T^{(k)}]$.

Since different nucleotide sites are assumed to evolve independently of each other (see Section 2.3), the conditional probabilities can be calculated site-by-site by iterating over the sequence alignment columns. Therefore, we will only examine the process of computing the conditional probabilities and the likelihood score for a specific site of the sequence alignment and will extend this to the whole alignment at the end of this section.

The individual entries of the conditional probabilities $\vec{L}^{(k)}$ are computed as follows: if the node k is a tip with state s, the conditional probabilities are:

$$L_{s_k}^{(k)} = \begin{cases} 0, & s_k \neq s \\ 1, & s_k = s. \end{cases} \tag{2.13}$$

A special case are sequence positions that contain ambiguous characters, i.e., characters that do not encode one of the four DNA bases A, C, G, T but multiple bases simultaneously. In this case, $L_{s_k}^{(k)}$ is set to 1 for all bases that are encoded by the ambiguous character. If k is an internal node, with descendants l and r and the corresponding branch lengths b_l and b_r (see

Figure 2.1 for a schematic representation), the conditional probability can be computed as

$$L_{s_k}^{(k)} = \left[\sum_{s_l=A}^{T} p_{s_k s_l}(b_l) L_{s_l}^{(l)}\right] \times \left[\sum_{s_r=A}^{T} p_{s_k s_r}(b_r) L_{s_r}^{(r)}\right]. \tag{2.14}$$

Since all descendant tips of node k are either descendants of its immediate child nodes l or r, the probability $L_{s_k}^{(k)}$ of observing the data at the descendant tips of k given s_k is equal to the probability of observing the data at the descendant tips of l given s_k times the probability of observing the data at the descendant tips of r given s_k. These are the terms within the two pairs of brackets of equation (2.14). The left bracket corresponds to child node l and the right bracket corresponds to r. Thus, the left term describes the probability $p_{s_k s_l}(b_l)$ that s_k will evolve into s_l over time b_l times the probability $L_{s_l}^{(l)}$ of observing the data at l's tips given the state s_l, summed over all possible states $s_l \in A, C, G, T$. The same applies to the right bracket accordingly.

In order to compute the likelihood score of the tree, one needs to compute the conditional probability vectors for all nodes of the tree. This can be done bottom-up from the tips to the root, i.e., by a post-order tree traversal.

In the following, we will use the sample tree depicted in Figure 2.2 to outline *Felsenstein's pruning algorithm*. The rectangles denote terminal nodes (numbered 1 to 5), while internal nodes (numbered 6 to 8) as well as the root (numbered 0) are represented by circles. The terminal nodes additionally show the associated species' nucleotide characters of an arbitrary site as given by the input matrix. The branch lengths $b_1..b_8$ represent the expected number of nucleotide substitutions per site and are fixed for the likelihood computation.

Node 1 and 2 are tips with state T and G, hence $\vec{L}^{(1)} = [0, 0, 0, 1]$ and $\vec{L}^{(2)} = [0, 0, 1, 0]$. This allows for computation of $\vec{L}^{(6)}$:

$$\begin{aligned} L_{s_6}^{(6)} &= \left[\sum_{s_1=A}^{T} p_{s_6 s_1}(b_1) L_{s_1}^{(1)}\right] \times \left[\sum_{s_2=A}^{T} p_{s_6 s_2}(b_2) L_{s_2}^{(2)}\right] \\ &= p_{s_6 T}(b_1) \times p_{s_6 G}(b_2) \end{aligned} \tag{2.15}$$

With $\vec{L}^{(3)} = [1, 0, 0, 0]$:

$$\begin{aligned} L_{s_7}^{(7)} &= \left[\sum_{s_6=A}^{T} p_{s_7 s_6}(b_6) L_{s_6}^{(6)}\right] \times \left[\sum_{s_3=A}^{T} p_{s_7 s_3}(b_3) L_{s_3}^{(3)}\right] \\ &= \left[\sum_{s_6=A}^{T} p_{s_7 s_6}(b_6) p_{s_6 T}(b_1) p_{s_6 G}(b_2)\right] \times p_{s_7 A}(b_3) \end{aligned} \tag{2.16}$$

$\vec{L}^{(8)}$ can be computed from $\vec{L}^{(4)} = [0, 1, 0, 0]$ and $\vec{L}^{(5)} = [0, 1, 0, 0]$:

$$\begin{aligned} L_{s_8}^{(8)} &= \left[\sum_{s_4=A}^{T} p_{s_8 s_4}(b_4) L_{s_4}^{(4)}\right] \times \left[\sum_{s_5=A}^{T} p_{s_8 s_5}(b_5) L_{s_5}^{(5)}\right] \\ &= p_{s_8 C}(b_4) \times p_{s_8 C}(b_5) \end{aligned} \tag{2.17}$$

And finally, the probability vector $\vec{L}(0)$ at the root can be computed with:

$$\begin{aligned} L_{s_0}^{(0)} &= \left[\sum_{s_7=A}^{T} p_{s_0 s_7}(b_7) L_{s_7}^{(7)}\right]_7 \times \left[\sum_{s_8=A}^{T} p_{s_0 s_8}(b_8) L_{s_8}^{(8)}\right]_8 \\ &= \left[\sum_{s_7=A}^{T} p_{s_0 s_7}(b_7) \left(\sum_{s_6=A}^{T} p_{s_7 s_6}(b_6) L_{s_6}^{(6)}\right)_6 \left(\sum_{s_3=A}^{T} p_{s_7 s_3}(b_3) L_{s_3}^{(3)}\right)_3\right]_7 \times \\ &\quad \left[\sum_{s_8=A}^{T} p_{s_0 s_8}(b_8) \left(\sum_{s_4=A}^{T} p_{s_8 s_4}(b_4) L_{s_4}^{(4)}\right)_4 \left(\sum_{s_5=A}^{T} p_{s_8 s_5}(b_5) L_{s_5}^{(5)}\right)_5\right]_8 \\ &= \left[\sum_{s_7=A}^{T} p_{s_0 s_7}(b_7) \left\{\sum_{s_6=A}^{T} p_{s_7 s_6}(b_6) \Big(p_{s_6 T}(b_1)\Big)_1 \Big(p_{s_6 G}(b_2)\Big)_2\right\}_6 \Big(p_{s_7 A}(b_3)\Big)_3\right]_7 \times \\ &\quad \left[\sum_{s_8=A}^{T} p_{s_0 s_8}(b_8) \Big(p_{s_8 C}(b_4)\Big)_4 \Big(p_{s_8 C}(b_5)\Big)_5\right]_8 \end{aligned} \tag{2.18}$$

For better readability, the terms have been grouped according to the tree structure. Round brackets indicate terminal nodes, whereas square brackets and the curly brackets describe internal nodes. The numerical subscripts correspond to the respective node numbers.

Once the conditional probability vectors for all nodes have been computed, the likelihood score for a specific site S can be computed at the root by means of:

$$f(S|\theta) = \sum_{s_0=A}^{T} \pi_{s_0} L_{s_0}^{(0)} \tag{2.19}$$

Here, θ incorporates the parameters of the respective model of sequence evolution as well as the branch lengths of the tree. For the computation of the likelihood score, θ is assumed to be fixed. π_{s_0} is the prior probability that the root has state s_0 as given by the base frequency of the specific nucleotide under the chosen model.

The overall likelihood score of the tree can be calculated by multiplying $f(S|\theta)$ over all m sites:

$$L = \prod_{S=0}^{m-1} f(S|\theta). \tag{2.20}$$

However, since all entries p_{ij} of the substitution probability matrix are ≤ 1, the individual $L_{s_k}^{(k)}$

values and consequently the $f(S|\theta)$'s can be very small. Hence, the log likelihood l is computed instead of L:

$$l = log(L) = \sum_{S=0}^{m-1} log(f(S|\theta)). \tag{2.21}$$

2.6.2 The Pulley Principle

As pointed out in Section 2.3.1, most evolutionary models utilize time-reversible Markov chains. Reversibility means that the Markov chain does not distinguish whether time is running forward or backward. With equation (2.4) this implies that

$$\pi_i p_{ij}(t) = \pi_j p_{ji}(t), \text{ for all } i \neq j. \tag{2.22}$$

Another feature of Markov chains is expressed by the Chapman–Kolmogorov equation which states that

$$p_{ij}(b_1 + b_2) = p_{ik}(b_1)p_{kj}(b_2). \tag{2.23}$$

This allows for an important property for the computation of the *phylogenetic likelihood function* which Felsenstein called the *pulley principle*. It allows for placing the root anywhere within the tree with no effect on the likelihood score of the tree. This is important for two reasons: firstly, it allows us to easily root any unrooted tree just by inserting a *virtual root* into any branch of the tree. Secondly, it allows us to maximize the likelihood score of the tree by optimizing its branch lengths (see Section 2.6.3.1 for more details).

For example, combining equations (2.18) and (2.19) results into:

$$f(S|\theta) = \sum_{s_0=A}^{T} \sum_{s_7=A}^{T} \sum_{s_8=A}^{T} \pi_{s_0} p_{s_0 s_7}(b_7) L_{s_7}^{(7)} p_{s_0 s_8}(b_8) L_{s_8}^{(8)}. \tag{2.24}$$

With equation (2.22) we can replace $\pi_{s_0} p_{s_0 s_7}(b_7)$ by $\pi_{s_7} p_{s_7 s_0}(b_7)$ and with equation (2.23) $\sum_{s_0} p_{s_7 s_0}(b_7) p_{s_0 s_8}(b_8) = p_{s_7 s_8}(b_7 + b_8)$ holds. Hence, we have

$$f(S|\theta) = \sum_{s_7=A}^{T} \sum_{s_8=A}^{T} \pi_{s_7} p_{s_7 s_8}(b_7 + b_8) L_{s_7}^{(7)} L_{s_8}^{(8)}. \tag{2.25}$$

This obviously corresponds to the likelihood score of the unrooted tree as depicted by Figure 2.3 which can be derived from the rooted counterpart of Figure 2.2 by removing the root at node 0 and merging the branches connecting node 7 and 8 to a single branch with length $b_7 + b_8$. Or, if looked at in the opposite direction, this allows for adding a *virtual root* to the unrooted tree by splitting the branch between nodes 7 and 8 and inserting a new node. In fact, adequate application of equations (2.22) and (2.23) shows that the *virtual root* can be placed on any branch of the tree.

Another interesting observation that can be made from equation (2.25) is the fact that the likelihood score only depends on b_7 and b_8 via their sum. As a consequence, equation (2.24)

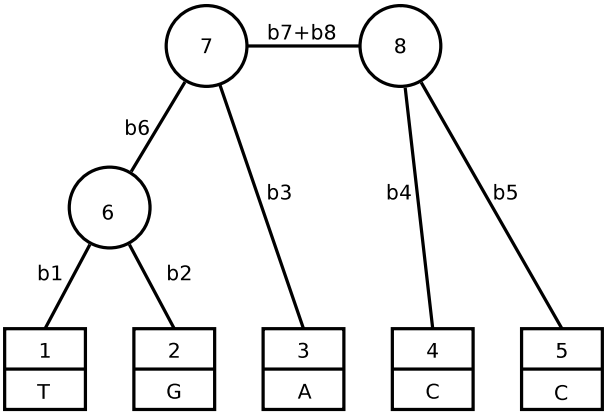

Figure 2.3: An unrooted 5 taxon tree

can be rewritten to

$$f(S|\theta) = \sum_{s_0=A}^{T} \sum_{s_7=A}^{T} \sum_{s_8=A}^{T} \pi_{s_0} p_{s_0 s_7}(b_7 + x) L_{s_7}^{(7)} p_{s_0 s_8}(b_8 - x) L_{s_8}^{(8)}. \tag{2.26}$$

Thus, the root can be placed anywhere on the branch between node 7 and node 8 and will always yield the same likelihood score as long as $b_7 + b_8$ remain constant. This is the reason why Felsenstein called it the *pulley principle*. The *virtual root* can be regarded as a pulley and all components to both sides of the pulley can be moved up and down without altering the likelihood score of the tree as long as they are moved by the same amount on both sides.

2.6.3 Optimization of the Likelihood Score

As already pointed out in the beginning of this section, the goal of phylogenetic tree inference under the ML criterion is to find the one phylogenetic tree with the highest likelihood score. The score, however, is not only influenced by the tree topology itself, but also by the branch lengths of the tree as well as the parameters of the evolutionary model. In that sense, both the branch lengths and the model parameters can be considered as parameters of the likelihood function. The Maximum Likelihood approach allows us to estimate these parameters so that the highest likelihood score for a given tree topology is achieved.

For the purpose of maximizing the likelihood score, the tree topology is assumed to be fixed and only the parameters are to be modified. Although the parameters of the likelihood function comprise the branch lengths ν and the model parameters ϕ, ν and ϕ are usually optimized separately for several reasons. Most importantly, the computational costs of the optimization increase with the number of parameters to be optimized. Since the branch lengths and the model parameters are not strongly correlated for most evolutionary models, a multivariate estimation of both is not necessarily required. Estimating ν and ϕ separately therefore saves computing time. Additionally, the branch lengths need to be optimized after each change to

the tree topology. As most programs for phylogenetic tree inference implement hill-climbing algorithms that frequently remove and re-insert subtrees, the branches need to be optimized very frequently. The model parameters on the other hand need to be optimized less frequently during the tree search: in the initial phase of the tree search they quickly converge to their optima and stay rather constant once a "reasonable" tree topology has has been established. Note, however, that typically not all branches of the tree are recomputed after topological modifications in most heuristics but only the branches of the surrounding nodes in order to reduce the computational effort.

2.6.3.1 Branch Length Optimization

Due to the numerical complexity of multivariate optimization techniques, the branch lengths are usually optimized one by one. The model parameters as well as all branches except the one branch to be optimized are assumed to be fixed. The optimization of the branch can then be performed by exploiting the *pulley principle*. It allows for placing a virtual root on any branch of the tree and guarantees that the likelihood score of the tree will only depend on the lengths of the branches leading to the virtual root.

If we consider the unrooted tree of Figure 2.3 and assume we want to optimize the length of the branch connecting node 7 with node 8, we would place the virtual root on that branch. Eventually, we would obtain the rooted tree of Figure 2.3 with node 0 being the virtual root. Yet, we would place the virtual root immediately besides node 8 such that $b_8 = 0$. As the likelihood score of the tree depends on b_7 and b_8 via their sum, it effectively only depends on b_7 as can be seen by inserting $b_8 = 0$ into equation (2.24). Generally, for any two nodes i and j and their connecting branch with length b, this equation can be rewritten to

$$f(S|\theta) = \sum_{s_i=A}^{T} \sum_{s_j=A}^{T} \pi_{s_i} p_{s_i s_j}(b) L_{s_i}^{(i)} L_{s_j}^{(j)}. \tag{2.27}$$

The likelihood score of the tree can then be optimized by altering b accordingly.

Most univariate optimization methods require at least the first partial and second partial derivatives of the objective function – in our case the likelihood function – with respect to the parameter to be optimized for finding the minimum and maximum values of the function. With equation (2.27) the first and second derivatives of l with respect to b can be calculated analytically (see [160]), which facilitates the application of many univariate methods for optimizing the branch lengths. In practice the *Newton-Raphson* method is often used for this task due to its favorable convergence properties. Appropriate algorithms are discussed in almost any textbook on numerical optimization (see [46] for example).

2.6.3.2 Model Parameter Optimization

The parameters of the evolutionary model that need to be estimated usually comprise the rate parameters of the substitution rate matrix ($a, ..., e$ for GTR, f is usually fixed to 1) and

optionally the parameters that account for rate heterogeneity among sites. In the case of a discrete-rates model, these are the rates r_k as well as the probabilities p_k. For the continuous as well as the discrete gamma-rates model only the shape parameter α needs to be estimated (p_k is fixed in case of the discrete gamma-rates model).

As the rate parameters are correlated they should be optimized simultaneously by means of a multivariate optimization method [161]. However, since no analytical solution for the partial derivatives of the likelihood function with respect to the rate parameters exists, they would have to be approximated by the difference method which is computationally very expensive. Though derivative-free optimization methods could be used instead, in practice they are not applied to this task as they impose similar computational effort but are considered to be less efficient [54]. Consequently, multivariate optimization methods like Newton-Raphson or Broyden-Fletcher-Goldfard-Shanno (BFGS) are typically not used for this task but univariate optimization techniques like Brent's method [14].

The α shape parameter of the gamma distribution is typically optimized with Brent's method, too. Regarding the discrete-rates model parameters, the same arguments as for the rate parameters and multivariate optimization apply. Hence, they are usually optimized individually with some univariate optimization method.

Note that, any change to the model parameters results in a modified substitution probability matrix and therefore requires all conditional probabilities at all inner nodes to be recomputed. Hence, a quickly converging optimization method is of paramount importance for an efficient implementation of the Maximum Likelihood approach.

2.7 Bootstrapping

Generally, nonparametric bootstrapping [32] is a method for assigning confidence values to statistical estimates. Felsenstein proposed to apply it to phylogenetic tree inference in order to obtain confidence limits on phylogenies [43]. Bootstrapping can be applied to any method for tree reconstruction, distance-based methods as well as character-based methods.

The method works as follows: A number of pseudo alignments, so-called *bootstrap samples* or *bootstrap replicates*, are generated by re-sampling sites from the original sequence alignment with replacement. The bootstrap samples have the same number of sites as the original alignment. The sites to be re-sampled are chosen randomly from the original alignment, so that some sites may be sampled multiple times while other sites may not be sampled at all. The bootstrap samples are then used in the same way as the original alignment to reconstruct phylogenetic trees, i.e., with the same algorithm and under the same evolutionary model. Afterwards, a consensus tree is built from the resulting bootstrap trees (see, e.g., [20] for a comprehensive review of consensus tree methods) that assigns support values to the individual clades of the phylogenetic tree resulting from the original alignment. A clade is a monophyletic group, i.e., a group of species consisting of a single common ancestor and all its descendants. A clade's support value is the proportion of bootstrap trees that include the clade.

While the process of bootstrapping and the computation of bootstrap support values is

straight-forward, the interpretation of the values is not. Often, the bootstrap supports are interpreted as a measure of the probability that the tree or clade is true, i.e., as a measure of accuracy. However, the only conclusion that can be drawn from bootstrapping is that the same tree or clade would be estimated from repeated sampling of the species' molecular sequences. This is the true nature of bootstrapping: instead of sequencing the species of interest multiple times and thereby creating multiple sequence alignments, we only re-sample from one single alignment and create pseudo alignments. In this sense, the bootstrap support values are a measure of repeatability. Yet, Hillis and Bull showed that under *certain conditions*, bootstrap proportions of over 70% usually correspond to a true clade [71] and consequently are a measure of accuracy. Many phylogeneticists have adopted the 70% value and use it as a rule of thumb. However, most of them neglect the "certain conditions" under which this value was obtained. One of the conditions is equal rate of change among sites which is an inappropriate assumption for most phylogenies as described in Section 2.3.3. In the same paper, Hillis and Bull point out that values over 50% are an overestimate of accuracy, if these conditions are not met. Hence, for most phylogenetic analyses (i.e., such that assume rate heterogeneity) a high bootstrap support does not allow for any conclusion about the correctness of the inferred tree topology. Bootstrapping does however provide "a systematic method of assessing the robustness of a data set to perturbation" as summarized by Sanderson [127].

Further discussion about the usefulness of bootstrap support values are clearly beyond the scope of this thesis. Nonetheless, most systematics journals require bootstrap support values on phylogenetic trees to be published. Consequently, the possibility to compute bootstrap support values is requested by most systematicists and has to be provided by any phylogenetic inference software if the inferred trees are to be published.

The actual implementation typically requires only little modification to existing programs as most of them already provide the possibility to assign weight values to individual sequence columns. These weights are multiplied with each site's likelihood score when computing the total likelihood score of the tree by means of equation (2.20). Due to the commutative property of the multiplication, the order of the sequence columns does not influence the tree's likelihood score. Re-sampling of the sequence alignment can therefore be easily implemented by assigning weights to the individual sites that correspond to the number of times the particular site has been sampled. For example, a site that has been sampled twice has a weight of 2, whereas a site that has not been sampled at all has a weight of 0.

Additionally to the question of how to interpret the bootstrap support values, another question that arises is "How many bootstrap replicates are enough ?". Computing one single bootstrap replicate requires approximately the same runtime as a normal tree inference. As normal tree inference under Maximum Likelihood already is a very compute intensive task, the number of bootstrap replicates that are being computed is often determined by the computing power and time available. The rule of thumb is that one typically requires 100-1000 bootstrap replicates, the more the better. Pattengale et al. [115] recently proposed two stopping criteria that allow for determining when enough bootstrap replicates have been generated at runtime. They assess their approach with 17 real-world DNA datasets and conclude that typically 100-500

replicates are sufficient.

Stamatakis et al. [133] developed a *rapid bootstrap algorithm* that is one order of magnitude faster than the standard algorithm described above. The speedup is achieved by application of heuristics that on the hand reduce the computational complexity of the likelihood function and on the other hand reduce the search space of tree inference. They assess their algorithm on 22 diverse DNA and amino acid datasets and show that the proposed heuristics yield bootstrap support values that are highly correlated to standard bootstrap algorithms but are between 8 and 20 times faster.

2.8 Bayesian Inference

In the mid-1990ies, several authors suggested to use Bayesian inference for reconstructing phylogenetic trees [91, 95, 120]. Although this thesis concentrates mainly on Maximum Likelihood based tree inference, the basic concepts of Bayesian inference will be discussed in this section as both approaches share some similarities. Consequently, many of the concepts for parallelizing the phylogenetic likelihood function that will be discussed in Chapter 5 can also be applied to Bayesian tools. An excellent discussion of Bayesian statistics, its application to phylogenetic inference, and a detailed description of the most commonly used algorithms can be found in [161].

Bayesian inference is based on Bayes' theorem which allows for computing the conditional probability $P(H|D)$ of a hypothesis H given data D as

$$P(H|D) = \frac{P(H)P(D|H)}{P(D)}. \tag{2.28}$$

Here $P(H)$ is the *prior probabilitiy* of H that is assigned to the hypothesis before the data are analysed. It may reflect additional prior knowledge on H or incorporate the subjective opinion of the user. $P(H|D)$ is called the *posterior probability* and describes the probability of the hypothesis after additional evidence (in this case the observed data D) has been taken into account. The conditional probability $P(D|H)$ is the *likelihood* of observing data D under the hypothesis H. $P(D)$ is the marginal probability of D and acts as a normalizing constant.

When modeling physical processes, uncertainties are typically modeled as random variables and the probabilities of the individual values they may take are described by a probability distribution. Bayes' theorem can be extended to probability distributions and then takes the following form for a random variable θ.

$$f(\theta|D) = \frac{f(\theta)f(D|\theta)}{f(D)}. \tag{2.29}$$

Accordingly, $f(\theta|D)$ is called the *posterior probability distribution*, $f(\theta)$ the *prior probability distribution*, and $f(D|\theta)$ the *likelihood function*. The marginal probability of the data $f(D)$ makes $f(\theta|D)$ integrate to 1. It describes the total probability of the data integrated over the

whole parameter space of θ. Hence, Equation (2.29) can be rewritten to

$$f(\theta|D) = \frac{f(\theta)f(D|\theta)}{\int f(\theta)f(D|\theta)d\theta}. \tag{2.30}$$

When applied to phylogenetic inference, the posterior probability of a phylogenetic tree τ_i can be computed as

$$f(\tau_i|D) = \frac{f(\tau_i)f(D|\tau_i)}{f(D)} = \frac{f(\tau_i)f(D|\tau_i)}{\sum_{j=1}^{T(n)} f(\tau_j)f(D|\tau_j)} \tag{2.31}$$

with $T(n)$ being the total number of possible tree topologies for n species (Equation (2.1)). Computing the likelihood $f(D|\tau_i)$ requires integration over all combinations of branch lengths ν and model parameters ϕ:

$$f(D|\tau_i) = \int_{\nu}\int_{\phi} f(D|\tau_i, \nu, \phi)f(\nu, \phi)d\nu d\phi \tag{2.32}$$

For a fixed tree topology τ_i and fixed branch lengths ν and model parameters ϕ, the likelihood $f(D|\tau_i, \nu, \phi)$ can be computed by means of the phylogenetic likelihood function (see Section 2.6.1). For the prior distributions, a uniform prior $f(\tau_i) = T(n)^{-1}$ is usually chosen for the tree probabilities and a uniform or exponential prior $f(\nu, \phi)$ for the branch lengths and model parameters.

As $T(n)$ grows factorially with n and due to the large parameter space of ν and ϕ, the summation and integrals in Equations (2.31) and (2.32) cannot be evaluated analytically. Therefore Markov chain Monte Carlo (MCMC) simulations are typically used in Bayesian inference to approximate the posterior probability distribution [98, 67, 60]. MCMC allows for taking (dependent) samples from a probability distribution of interest. It uses a Markov process with a steady-state distribution that is equivalent to the probability distribution. For phylogenetic inference, any combination of a tree topology τ, a vector of branch lengths ν and model parameters ϕ encodes a particular state of the Markov chain. In each iteration of the Monte Carlo simulation, a state change of the chain, i.e., a modification to the tree topology (by means of, e.g., NNI or SPR moves), the branch lengths, or the model parameters, will be proposed. The proposal will typically be accepted if it improves the likelihood score of the tree, but "bad" modifications will also be accepted randomly with a certain probability. Every k-th iteration, a sample of the Markov chain will be taken. If the simulation runs long enough, the proportions of the individual states in the samples will correspond to the steady-state distribution of the Markov chain and hence the posterior probability distribution.

If the target distribution has multiple peaks, the Markov chain may have difficulties in moving from one peak to another. Consequently, it may get stuck on one peak and the resulting samples will not approximate the posterior probability distribution correctly. This is similar to the problem of ML-based hill-climbing algorithms that can get stuck in local maxima and therefore miss *the* Maximum Likelihood tree. To avoid this problem, a variant of MCMC is typically used for phylogenetic inference: the Metropolis-coupled MCMC (MCMCMC or

MC^3) algorithm [53]. In MC^3, multiple chains with different stationary distributions are run in parallel. The first chain is called the *cold chain*, the other chains are the *hot chains*. Only the cold chain approximates the correct posterior probability distribution, whereas the hot chains apply more aggressive state changes in order to move more easily from one peak of the probability distribution to another. Occasionally, states are swapped between two randomly chosen chains (which also includes the cold chain). Hence, if the cold chain is stuck in one peak, it may leave this peak when it swaps states with one of the hot chains.

The main advantage of Bayesian inference over ML-based inference of phylogenetic trees is the fact that, the posterior probability of a phylogenetic tree is easy to interpret: it is the probability that the tree is correct given the data, model and prior. In contrast to that, Maximum Likelihood based approaches require the computation of multiple bootstrap replicates in order to obtain some sort of confidence measure which even is not that clear to interpret (see Section 2.7). Another benefit of Bayesian methods for phylogenetic tree inference is the fact, that the branch lengths and parameters of the utilized evolutionary model do not need to be optimized explicitly but are sampled during the simulation. In ML-based inference, on the other hand, they need to be optimized separately by means of computationally expensive iterative optimization methods like Newton-Raphson or Brent's algorithm (see Sections 2.6.3.1 and 2.6.3.2).

However, Bayesian inference also has drawbacks. The main problem is, that it is difficult to decide whether the simulation has been run long enough for the Markov chain to reach its stationary distribution. Hence, it is not clear whether the approximated posterior distribution of the phylogenetic tree is correct or not.

2.9 Programs for Phylogenetic Tree Inference

This section describes a selection of state-of-the-art applications for phylogenetic tree inference, their features, and which type of parallelism as outlined in Section 5 they exploit (if any).

2.9.1 RAxML

RAxML (*R*andomized *Ax*elerated *M*aximum *L*ikelihood) is an ML-based program developed by Alexandros Stamatakis [135]. It has originally been derived from fastDNAml by Olsen et al. [106] which in turn has been derived from DNAml that is included in Felsenstein's PHYLIP (see next section). It performs a heuristic tree search that is based on a "classical" hill-climbing algorithm, i.e., the likelihood score of the tree is gradually optimized by performing rearrangement operations on the tree topology like SPR (Subtree Pruning and Re-grafting) moves. *RAxML* also implements a Maximum Parsimony component that is, however, only used for creating "good" starting trees that are then further optimized based on the Maximum Likelihood criterion.

Besides steady improvements of the search algorithm, a special focus during the development of *RAxML* has always been on technical optimizations in order to improve the performance of the program. For example, the computationally most expensive parts of the phylogenetic likelihood

function have recently been implemented as hand-optimized SSE3 assembler code that is up to 50% faster than compiler-generated code [10]. In general, *RAxML* is very memory-efficient and typically requires significantly less memory than other state-of-the-art programs for phylogenetic inference like GARLI, IQPNNI, or PHYML [131]. In addition to that, it typically yields better trees (in terms of the likelihood score) in shorter time than those programs [131, 137].

RAxML supports DNA, protein, binary, multi-state, and RNA secondary structure sequences as input data. The input sequences can be divided into multiple partitions to each of which an individual model of sequence evolution along with its private set of model parameters can be assigned. Each partition can be composed of a different data type, e.g., it is possible to mix nucleotide and protein sequences. *RAxML* is able to perform standard bootstrap analyses as described in Section 2.7 as well as so-called rapid bootstrap analyses which are up to one order of magnitude faster than the standard approach [133]. Additionally it implements the WC and FC Bootstrap convergence criteria [115] that allow for determining automatically whether enough bootstrap replicates have been computed.

The most recent version as of writing this thesis is 7.2.6 and is provided as open source. It comprises a sequential version, a PThreads-based version that exploits loop-level parallelism (see Section 5.2.3), and an MPI-based version that exploits embarrassing parallelism (see Section 5.2.1). It can additionally be compiled as a hybrid PThreads/MPI application that exploits loop-level and embarrassing parallelism simultaneously. Besides that, various other parallelizations approaches of *RAxML* have been implemented over time. A comprehensive overview of those is provided in Section 5.3.

All concepts for the parallelization of phylogenetic tree inference algorithms discussed in this thesis have been implemented in *RAxML*.

2.9.2 PHYLIP

The *Phyl*ogeny *I*nference *P*ackage *PHYLIP* [44] is developed by Joseph Felsenstein since 1980 and is probably the most widely distributed tool suite for phylogenetic tree inference. It is freely available in source code, but also distributed as pre-compiled executables for various operating systems. *PHYLIP* is not an unified program but a collection of 35 distinct programs, each of which performs a specific task that is required for phylogenetic analyses. The outputs and inputs of the particular programs can be connected such that a toolchain can be built that fits the requirements of a specific analysis. *PHYLIP* provides tools for inferring phylogenetic trees using various distance methods, Maximum Parsimony, and Maximum Likelihood. It is able to process nucleotide and protein sequences as input data, as well as gene frequencies, restriction sites, and discrete characters. Additionally, it allows for bootstrap analyses and the computation of consensus trees and also provides tools for printing and manipulating trees.

Although *PHYLIP* is widely distributed, it is typically not used for (large-scale) real-world analyses. This is due to the fact, that its actual implementation is not optimized with regard to efficiency and performance but can be considered as a proof-of-concept implementation. Additionally, all tools of the package are sequential which prohibits them from being used for

large-scale analysis. Nevertheless, it is fully functional and the tools that do not implement computationally expensive methods can be used as swiss army knife for phylogenetics. Additionally, it serves as reference for other applications and for cross-checking results.

2.9.3 GARLI

GARLI (*G*enetic *A*lgorithm for *R*apid *L*ikelihood *I*nference) is a tool for phylogenetic tree inference under the Maximum Likelihood criterion developed by Derrick Zwickl [163]. It employs a stochastic genetic algorithm for finding the tree topology, branch lengths and model parameters that maximize the likelihood score.

In general, genetic algorithms work as follows: a population of candidate solutions to an optimization problem – so-called individuals – evolve toward better solutions from generation to generation. In each generation, the fitness of every individual is evaluated according to some scoring function and a fraction of the best scoring individuals are randomly selected. The parameters of the selected individuals are then recombined and/or randomly mutated to form a new generation of the population that is used for the next iteration of the algorithm. The algorithm terminates when a satisfactory fitness level has been reached or after a maximum number of iterations have been produced.

In *GARLI*, an individual is defined as a tree topology along with its branch lengths and the parameters of the evolutionary model utilized for the tree search. Each individual is assigned a fitness level based on the likelihood score of its tree. In each generation, random mutations, which can either be topological mutations, branch length mutations, or model parameter mutations, are applied to the best scoring individuals. Topological mutations consist of tree rearrangement moves like NNI (Nearest Neighbor Interchange) or SPR. In case of branch length and model parameter mutations, some branches/model parameters are chosen and their current values are multiplied by gamma-distributed random variables. The mutated individuals then constitute the next generation of the population. When the fitness levels of the population do not improve for a certain number of generations, the tree search terminates.

GARLI supports DNA, protein, and codon sequence alignments as input data and is able to perform multiple tree searches and bootstraps in a single program run. An OpenMP-based version that exploits loop-level parallelism as well as an MPI-based version that exploits embarrassing parallelism are available. The most recent version is 1.0 and can be downloaded in source code [1].

2.9.4 MrBayes

MrBayes by John Huelsenbeck and Fredrik Ronquist [73, 123] is probably the most widely used program for Bayesian inference of phylogenetic trees. It employs a Metropolis-coupled Markov chain Monte Carlo (MC^3) algorithm as described in Section 2.8 for efficiently estimating posterior probability distributions of phylogenetic trees .

MrBayes is able to process DNA, protein, restriction site, and morphological data as input sequences. It supports mixed datasets, i.e., it allows for analyzing input datasets that comprise

multiple partitions of different data types.

The most recent version is 3.1 and is provided as open-source. It can be compiled as sequential application as well as MPI application. The MPI version exploits inference parallelism and distributes the individual chains of the MC^3 algorithm over multiple processors or nodes [7].

2.9.5 PAUP*

*PAUP** (*P*hylogenetic *A*nalysis *U*sing *P*arsimony (*and Other Methods)) is a commercial tool developed by David Swofford and distributed by Sinauer Associates [144]. According to the author, it is "the most widely used software package for the inference of evolutionary trees". As the name suggests, it has originally been designed as a parsimony tool, but the most recent version 4.0 adds support for Maximum Likelihood and distance methods.

DNA and protein data are supported as input sequences as well as mixed datasets for partitioned analyses. *PAUP** is able to perform non-parametric bootstrap analyses and jackknife resampling analyses [32] for assigning measures of confidence to inferred tree topologies. Jackknife resampling is essentially very similar to bootstrapping, but instead of resampling to a dataset of the same size as the original dataset, only a subset of the original dataset is used for the analysis. Besides a heuristic tree search algorithm, *PAUP** also implements an exact search algorithm that guarantees to find the optimal tree under the respective optimality criterion. However, due to the previously described factorial growth of the tree search space (see Section 2.2) the exact algorithm is probably hardly used for real-world phylogenetic analyses.

*PAUP** is generally distributed as binary executable for MacOS, Windows, and DOS, but a portable source-code version is also available upon request. The MacOS version provides a feature-rich graphical user interface, whereas all other versions are command-line driven. Currently, no parallel version of *PAUP** is available.

Chapter 3

Shared Memory Multiprocessors

This chapter gives an overview of (multi-core) shared memory architectures, their memory subsystems, and suitable programming paradigms for these architectures. Additionally, it pinpoints potential performance bottlenecks of such architectures and provides concepts for exploiting their performance efficiently. Finally, it describes the `autopin` tool which implements many of these concepts and allows for automated performance-aware thread placement on multi-core systems.

3.1 Introduction

Shared memory multiprocessors are the prevalent architecture for small and mid-range parallel computers. They typically consist of a few to a couple of hundred processors and the main memory of these systems is shared between all processors – hence the name. Consequently, the processors also share the same physical address space. In fact, *shared address space multiprocessor* would be a more accurate term to describe these systems as they do not necessarily have a single global memory that is shared by all processors. Instead each processor may have its own memory that is indeed referenced by unique global addresses but can only be accessed by others via the processor to which the memory is actually connected to (see Section 3.3.3 for more details). Communication between the individual processors of a shared memory system occurs implicitly by performing read and write operations from/to the shared memory.

Before the introduction of multi-core processors, such architectures were only available in high performance computing (HPC) environment and for enterprise servers. However, with the ubiquity of multi-core processors, all computers, ranging from notebooks over desktop machines and workstations to large servers, are shared memory multiprocessors. In his widely cited article "The free lunch is over" [143] Herb Sutter claimed in 2005 that a "concurrency revolution" had already started. In fact, a clear paradigm shift is taking place in computer architecture with dramatic implications on software design. Parallel programming is leaving the high performance computing niche and consequently most software developers are now facing problems and encountering challenges the HPC community has been dealing with for decades. But even for veterans in parallel programming the new processor architectures exhibit a plethora

of new properties and features that have to be accounted for in order to efficiently exploit their performance.

The terminology used in the context of shared memory systems is not clearly defined and often even ambiguous. For example, the abbreviation *SMP* is often used for *symmetric multiprocessor* systems. However, it is not clear to which component or property of the multiprocessor the adjective "symmetric" refers to. Sometimes the same abbreviation is also used for *shared memory multiprocessor* systems. In fact, Hennessy and Patterson even use both terms for the very same abbreviation in two books [69, 116]. Therefore, it is important to define the respective terms properly before usage: in the following the term *multi-core system* will be used for systems equipped with a single multi-core processor and *multi-socket system* for "traditional" shared memory multiprocessor architectures with two or more processor sockets, each of which can be equipped with single- or multi-core processors. Where no distinction is required, the umbrella term *shared memory multiprocessor system* will be used for both.

3.2 Multi- and Many-Cores

3.2.1 The Power Wall

In the past, the ever increasing performance of microprocessors was mainly achieved through two techniques: raising the processor's clock rate and increasing the number of transistors. Higher clock rates result in higher performance without any modifications of the processor's microarchitecture and thus without the need to re-compile or even modify applications to exploit the increased performance resources. Yet, high clock rates have significant implications on the power consumption of a microprocessor. In general, the power consumption p of a processor can be approximated by

$$p = c \cdot v^2 \cdot f. \tag{3.1}$$

with c being the capacitive load, v the voltage, and f the switching frequency of the transistors [153]. The capacitive load mainly depends on the number of transistors and the technology used, whereas the switching frequency is a function of the clock rate. Compared to the voltage, which contributes to the processor's power consumption quadratically, the influence of the clock rate does not seem to be severe at first sight. However, increasing the clock rate and thus the switching frequency of the transistors also requires increasing the voltage in order to allow for stable operation. The clock rate therefore is the parameter with the highest impact on the power consumption of a microprocessor.

Although power consumption is critical for mobile computers that run on battery, for workstations and servers problems do not arise from the power consumption itself but from the associated heat dissipation. For desktop computers and workstations the heat can be dissipated by airflow. Though not very costly or technically challenging, the cooling fans required for this task induce noise which can be annoying for the user in an office environment. In large server installations, however, more elaborate and expensive cooling techniques are required to dissipate the heat produced by the microprocessors. Typically the same amount of energy is

required for cooling the computer system, that has previously been fed into the processor in form of electrical energy. With steadily increasing prices for electrical power, the operational costs over a server's lifetime can easily exceed its purchase price.

At some point – for which the expression *power wall* has been coined – the power consumption does not allow for increasing the clock rate any higher for both, economical as well as ecological reasons. A prominent example is Intel's Pentium 4 processor series that reached an all-time high at 3.8 GHz with the introduction of the Prescott core (based on the Netburst microarchitecture). The thermal design power (TDP) of this processor has been specified with 115W which had to be dissipated by the processor's die surface of only 109 mm^2. Tejas, the planned successor to the Prescott processor, had been canceled by Intel as it would have reached TDP values of over 150W.

3.2.2 From Single- to Multi-Core

Since the power wall prevents the microprocessor manufacturers from steadily increasing the computational power by increasing the clock rate, they have to concentrate on the second technique: increasing the number of transistors. Fortunately, Moore's law, which states, that the number of transistors in a microprocessor doubles every 18-24 months, still holds. Before the power wall had been hit, these additional transistors have mainly been used to increase the processor's cache and to increase the microarchitecture's complexity by adding more – either specialized or redundant – functional units to the processor core. Though bigger caches can increase the performance of many applications by buffering accesses to frequently used memory regions, they do not increase the processor's performance itself. They only alleviate the impact of slow accesses to the main memory, the performance of which grew at a substantially lower rate in the past years than the performance of the microprocessors.

Having several independent functional units within a processor core allows for parallel execution of multiple instructions, so-called *instruction level parallelism* (ILP). The processor exploits ILP transparently, i.e., the concurrent execution of machine instructions is hidden from the programmer who needs to just provide sequential code. This requires elaborate techniques like pipelining, out-of-order execution, multiple-issue, dynamic scheduling, and speculative execution to name but a few. These techniques increase the microarchitecture's complexity tremendously as they induce additional administration overhead. Furthermore, data as well as control dependencies that cannot be resolved by the microprocessor often prevent parallel execution. However, data and control dependencies can often be detected during compile time and hence the compiler can reorganize the resulting machine code in order to facilitate ILP.

Besides exploiting instruction-level parallelism, redundant functional units in a microprocessor can also be used to exploit *data level parallelism*, i.e., to execute the same operation on multiple input values concurrently. While it is difficult for the processor to detect and exploit data level parallelism automatically, it is typically much easier for the compiler. For example, adding two vectors of length n requires n add operations that are independent from each other and can therefore be executed in parallel. As this computation is typically implemented as a

loop over the vectors, the individual iterations can be easily identified by a compiler to be data parallel and hence be marked for parallel execution. For that purpose, all superscalar microprocessors provide SIMD (single instruction multiple data) instructions that operate on vectors instead of scalars. The best known examples for SIMD instruction sets are the MMX and SSE extensions of Intel's and AMD's x86 processors and the AltiVec instructions as implemented in PowerPC processors. These instructions typically operate on vectors of two or four integer or floating point values and hence allow for parallelization at a very fine grained level. In case of the above vector example, two iterations of the add loop can be fused into a single SIMD operation and be executed concurrently. Since the data and control dependencies are resolved automatically by the compiler this approach requires no additional circuits within the processor and no interaction by the programmer.

Although parallelism can be hidden from the programmer up to a certain degree by the techniques discussed above, many algorithms and programs exhibit a more coarse grained type of parallelism that requires different approaches in order to be exploited. Server applications for example have natural parallelism among the queries that are submitted by client applications. These queries are usually independent from each other and can therefore be processed concurrently. This higher level parallelism is known as *thread level parallelism* (TLP) as it can be processed in separate threads of execution. Note that the term "thread" in this context does not necessarily correspond to the concept of a thread in operating systems terminology. Each thread of execution can either be executed as a separate process or as a separate thread (i.e., a light-weight process).

TLP can be used to increase the utilization of the functional units of a superscalar microprocessor: *simultaneous multithreading* (SMT) allows for interleaving the individual instructions of multiple threads into a single stream that can be processed in parallel by applying the techniques for exploiting ILP listed above. The processor presents itself to the operating system as a multiprocessor (with two or more logical processors) which enables the operating system to schedule threads or processes on the logical processors for parallel execution like on any other (real) multiprocessor system. Since the instructions of one thread are independent from any other thread's instructions, SMT can help to keep the processor's pipeline filled when it would stall otherwise, e.g., in case of unresolvable data or control dependencies, or while waiting for memory accesses or I/O operation to finish. Intel introduced its SMT implementation *Hyper-Threading Technology* (HTT) in 2003 with the Pentium 4 and claimed that HTT increases the performance of many applications up to 30% but only requires 5% of the processor's die size [93].

Although SMT can obviously increase the performance of microprocessors at little cost, it is only a tool to keep the complex superscalar processor core busy. The straight-forward approach for exploiting TLP certainly is executing the individual threads in parallel on multiple processors. However, while multiprocessor systems are common in HPC and for enterprise server infrastructures, the desktop and workstation market has been dominated by single processor machines. Yet, this changed with the introduction of chip multiprocessors (CMP), which are better known as multi-core processors: the steadily increasing number of transistors available on a processor die (thanks to Moore's law) not only allows for increasing the number of functional

units within the processor core but also for increasing the number of processor cores. Mainly encouraged by power constraints, the microprocessor manufacturers switched from continuously increasing the complexity of the single processor core to actually lowering its complexity and compensating for the thereby induced performance loss by duplicating it. Intel for example, abandoned the complex Netburst microarchitecture in 2006 and switched to the Core microarchitecture which is based on Netburst's predecessor, the P6 microarchitecture that has been introduced in 1995 with the Pentium Pro. The first processor that was based on the Core microarchitecture shipped as a dual-core processor with clock rates up to 2.33 GHz and a TDP of 31 W (codename Yonah) [78]. While the Netburst-based Pentium 4 processor is specified with a theoretical peak performance of 7.60 GFLOPS (at 3.8 GHz) [79], a single Yonah core at 2.33 GHz only achieves 3.49 GFLOPS. However, the combined performance of both cores amounts to 6.99 GFLOPS which is almost on par with the Pentium 4 – but at a quarter of the Pentium's power consumption.

CMPs allow for designing power-efficient but yet powerful microprocessors. The downside of this approach is that applications cannot exploit the performance without modifications to their source code. A sequential program will only run on a single core and thus only exploit a fraction of the processor's performance. In fact, since the individual core of a CMP is less powerful and operates at a lower clock rate than the previous single-core processors, an unmodified program will typically run slower on a new multi-core processor model than it used to on an older single-core processor. In order to fully exploit the potential of a multi-core processor, applications have to be parallelized at thread level. As multi-core processors are conceptually very similar to shared memory multiprocessor systems, the same programming techniques that have been developed over decades for these systems can also be applied for exploiting parallelism on multi-core systems. The corresponding programming paradigms and approaches will be discussed in Sections 3.4 and 3.5.

3.3 The Memory Subsystem

Although the main properties that are used for describing the performance of a computer are typically the processor model, its clock rate, and the number of cores, the characteristics of its memory subsystem are similarly important for the overall system performance. In fact, particular properties of the memory subsystem can have significant impact on the performance of an application if they are not accounted for.

3.3.1 The Memory Wall

Unfortunately, the performance of the DRAM-based main memory did not keep pace with the microprocessor's computational capabilities. While Moore's law still holds, the performance of CPUs still doubles every 18 month. However, this is not the case for DRAM memory which only experiences performance increases of approximately 10% per year.

This problem is known under the term *memory wall* and is probably illustrated best by an

example: The Woodcrest and Clovertown processors were the first dual- respectively quad-core processors that implemented the Intel Core2 microarchitecture. Each core of these processors is able to perform four single precision floating point operations per cycle. As the clock-rate of the fastest Woodcrest and Clovertown processor models (Xeon 5160 and Xeon X5365) is 3.00 GHz, their theoretical peak performance is 24 GFLOPS and 48 GFLOPS respectively. The Core2 microarchitecture utilizes the so-called front-side bus (FSB) for connecting to I/O devices as well as to the memory. Both processors allow for a maximum FSB bandwidth of approximately 10.2 GB/s which can also be assumed as the maximum memory bandwidth. Consequently, the dual-core Woodcrest processor achieves a Bytes/FLOP ratio of 0.44, i.e., every floating point operation can access 0.44 bytes of operands from main memory on average. However, as a single floating point operand is 32 bits or 4 bytes wide, this implies that only every 10th operation can read a new operand from the main memory. Consequently, the performance of the main memory limits the processor to 10% of its theoretical peak performance. For the quad-core Clovertown processor the Bytes/FLOP ratio even drops to 0.22 which results in a potential performance loss of 95%.

3.3.2 Memory Hierarchies

In order to alleviate the impact of the memory wall, memory accesses are not processed directly by the main memory but by a series of fast but small caches: when the processor needs to read from main memory, it first checks whether the requested data word is contained in one of the caches. If so, a *cache hit* has occurred and the processor reads from the cache instead of the main memory which is significantly faster – both bandwidth-wise as well as latency-wise. If the word is not found in any of the caches, a *cache miss* has occurred and the word is read from the main memory. However, a copy of the word is additionally stored in at least one of the caches. This allows for exploitation of temporal locality: data that has been accessed once is very likely to be accessed again very soon. Future accesses to the same word can then be handled by the cache quickly. However, as the size of a cache is limited, new data words can only be stored in the cache if others are evicted at the same time. Since future memory accesses are hardly predictable in hardware and in order to facilitate the exploitation of temporal locality, most microprocessors employ a LRU (least recently used) strategy for replacing data in the cache. That is, the data that have not been accessed the longest, are the next to be evicted from the cache.

For efficiency reasons, caches work on a larger granularity than single data words, so-called cache lines which are typically between 32 and 64 bytes long. If a cache miss occurs, the cache controller will not only fetch and store the requested word but a whole cache line, i.e., a 32 or 64 bytes block that contains the requested word. As reading a larger block from main memory (so-called burst read) has approximately the same cost as reading a single word, this induces no penalty. Additionally, the principle of spatial locality suggests that if a data location is accessed, it is very likely that consecutive accesses will address nearby memory locations. If these memory locations belong to the same cache line as the previous access the corresponding

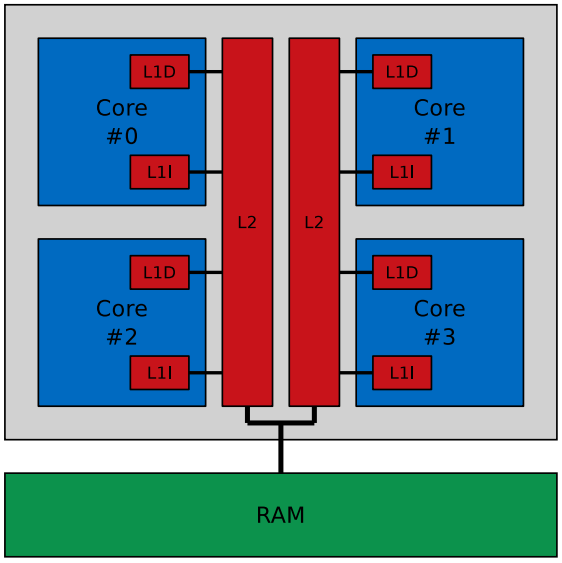

Figure 3.1: A quad-core processor with the L2 caches being shared between each two cores.

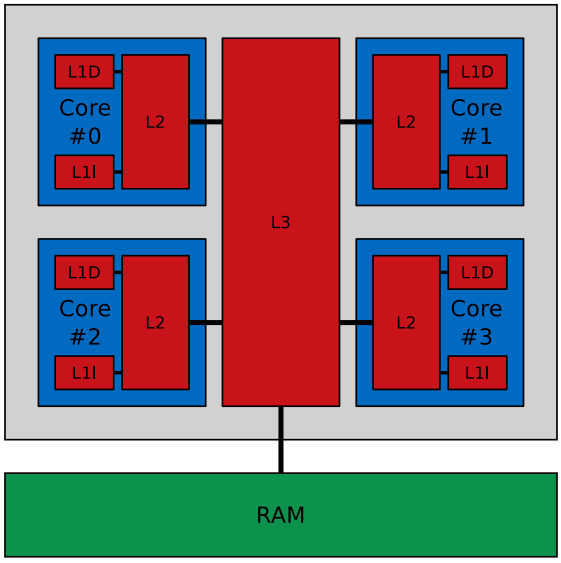

Figure 3.2: A quad-core processor with separate L2 caches and shared L3 cache.

data can be read from the cache and hence an additional memory access can be avoided.

Memory writes are also performed at cache line granularity. This implies that a single word can not be written to the main memory directly. Instead, the cache line to which the word belongs to will be read, modified according to the write operation, and stored in the cache. Depending on the cache policy, the modified cache line may be written back to the main memory immediately (*write through*) or only when it has to be evicted from the cache due to another cache miss (*write back*).

As fast caches are expensive to implement, there exist several levels of caches. The general rule of thumb on the size and speed of caches is as follows: the cache at the lowest level is small and fast, the caches at higher levels are bigger and slower. Cache misses at a lower level will be propagated through the cache hierarchy. Processor registers are cached by the level 1 data cache (L1D), microinstructions by the level 1 instruction cache (L1I). The distinction between data and instructions at this cache level is necessary in order to ensure that the program code is not evicted from cache due to data reads. Both L1 caches are typically a few kilobytes large and are cached by the level 2 cache (L2) which is usually unified (i.e., the cache is shared by data and instructions) and between 256 kB and 2 MB in size. On some architectures, there is an additional level 3 cache (L3) of a few megabytes. The last level cache (i.e., the L2 or L3 cache depending on the architecture) eventually connects to the main memory.

While the memory hierarchies of single-core processors used to be very homogeneous and their properties easy to understand, they become significantly more complex on multi-core systems: Depending on the architecture, some levels of the cache hierarchy may be shared by several cores while others are not. For example, when Intel introduced the first dual-core processors of the Core architecture, each core had its own L1 caches but the L2 was shared by

both cores. The cache hierarchy of the Core architecture's quad-core processors [80], however, is more complicated: as these processors are no native quad-cores but essentially two dual-cores in the same processor package (a so-called multi-chip module), only two cores share a common L2 cache (see Figure 3.1 for a schematic representation). AMD took a slightly different approach for its multi-core processors. AMD's first dual-core processor was not a native multi-core processor but – similarly to Intel's first quad-core processor – two single cores of the K8 microarchitecture assembled in one processor package [4]. Consequently, there is no shared last level cache on these processors but both cores have separate L1 and L2 caches. Yet, AMD introduced a shared L3 cache with the K10 multi-core microarchitecture [6]. Intel followed the same path with the introduction of their native quad-processors that implement the Nehalem microarchitecture [82]. A schematic representation of both architectures is provided in Figure 3.2.

The combination of separate caches and shared memory in multi-core processors induces a problem that is well-known for multi-processor systems: cache coherence. A cache line may not only be stored in main memory, but additional copies of it may reside in several caches. This certainly creates no problems on single-core/single-processor systems since only one processor can perform write operations and therefore can easily keep the copies in the caches coherent. However, on systems with more than one processor, cache coherence is much harder to achieve. It is always possible that one processor writes to a memory block for which another processor holds a copy in its cache. If this copy is not updated or at least invalidated (i.e., marked as outdated and hence invalid) upon write, subsequent read operations that are fulfilled by the outdated copy will deliver inconsistent data. Cache coherence protocols guarantee that all processors of a system have the same view of the main memory, i.e., that no cache delivers outdated data upon read. Smaller shared memory systems usually employ snooping protocols for keeping caches coherent, whereas larger systems typically use directory-based protocols for better scalability. In case of snooping protocols, each processor observes the memory bus for write transactions that affect cache lines for which it holds copies and updates its caches accordingly. In larger systems that typically implement non-uniform memory architectures, a global directory keeps track of the contents of all caches. Upon a write transaction the directory either updates or invalidates all caches that hold a copy of the respective cache line. For a detailed discussion of caches and cache coherence protocols see [65].

3.3.3 UMA and NUMA Architectures

Shared memory multiprocessors can generally be divided into *uniform memory access* (UMA) and *non-uniform memory access* (NUMA) systems. In UMA systems, the costs for accessing the main memory, i.e., the latency and available bandwidth, are identical for all processors and memory addresses. This is typically the case for systems where all processors are connected to the main memory via a shared bus. But UMA also implies that the memory bandwidth is shared between all processors which can become a bottleneck for large systems. Larger systems are therefore mostly based on the NUMA architecture that can overcome this problem as it does not imply a single global main memory and bus. Instead, each processor has its

own, directly attached, local memory which can be accessed at a high bandwidth and with low latency. The address space, however, is global and the local memory of each processor can be remotely accessed by all other processors. Contrary to UMA architectures, adding an additional CPU to the system increases the aggregate memory bandwidth and therefore allows for better scalability. Yet, remote memory accesses certainly induce higher latencies and the high bandwidth can only be utilized if the memory access is scattered over all processors.

As multi-core processors have become the de-facto standard for powerful CPUs, even the smallest multi-processor systems with only two sockets need to implement a NUMA architecture in order to provide enough memory bandwidth to all cores. What used to be a a rather small system with only two processors, now has up to eight cores in total if equipped with quad-core processors. Systems with similar core counts were considered as mid-range in the single-core era and would therefore presumably be designed as NUMA systems for good reasons. For x86-based systems, NUMA became the standard for multi-processor systems with the introduction of Intel's new Nehalem architecture in 2009 at the latest. The Frontside Bus based UMA architecture used by Intel's previous multi-core processors has been discarded and replaced by the new point-to-point QuickPath Interconnect (QPI). The memory controller has been moved from the north bridge to the CPU into the so-called "uncore". AMD introduced a similar architecture already in 2003 with the Opteron processor series: the memory controller has been integrated into the processor and the point-to-point HyperTransport interconnect allows for connecting to other CPUs and I/O hardware.

3.4 Shared Memory Programming

The building primitive for shared memory programming are threads, which are provided by all modern operating systems. By concept, all threads of a single process have a common virtual address space and hence share a memory region. Processes are usually not allowed to access the address space of any other process. Though there are techniques available in many operating systems that allow for designating a memory region to be shared between several processes, the term "shared memory programming" usually only applies to multi-threaded applications. Communication between the individual threads occurs implicitly by writing and reading shared variables.

The main benefit of the shared memory programming paradigm is that all threads share the same address space. Hence, all data structures of an application are accessible to all threads and therefore no explicit data exchange between the individual threads is necessary. This is especially useful for exploiting data level parallelism as the programmer does not need to take care of distributing data chunks to the processing threads as is necessary for distributed memory machines. Instead, the threads can fetch the required data from the shared memory and also write the results of their computations back themselves without communication.

Dealing with threads directly can be very cumbersome for the programmer. Creating, managing, synchronizing, and destroying the individual threads can become complex and is extremely error-prone. OpenMP automates these tasks and provides a convenient application

programming interface that allows for easy exploitation of thread level parallelism. It extents the C/C++ and Fortran programming languages by a set of compiler directives and also provides a couple of library routines. It can be regarded as an infrastructure for semi-automatic program parallelization: the programmer only has to identify and mark the program regions that can be executed in parallel. The actual parallelization task like creating and synchronizing threads is carried out by the OpenMP-enabled compiler. Especially loops like those of the vector add example of the previous section can be parallelized easily just by prepending the respective OpenMP compiler directive to the loop body. The individual iterations of the loop are then distributed over several threads and executed in parallel. This concept is often referred to as the fork-join paradigm, as new threads forked as soon as a parallel region is entered, and are joined again at the end of the region.

OpenMP is also very popular for parallelizing existing applications as it does not require major code reorganization. It rather allows for concentrating on the computationally most expensive regions that can be parallelized with little effort. For example, many applications from the scientific domain often utilize computationally expensive iterative methods that spend most of the computing time within a rather small number of loops. Good parallel performance for such applications can therefore be achieved by only parallelizing the execution of those loops. Although a higher degree of parallel efficiency might be possible with a holistic parallelization, this approach is usually very cost-effective in terms of required programming effort.

Unfortunately, the shared memory programming paradigm loses much of its beauty on NUMA systems: in order to benefit from the higher performance of the local memory, the programmer has to devise appropriate data partitioning schemes that ensure that at least the most frequently accessed data structures are stored in the local memory. As this can be very cumbersome to achieve with most shared memory programming approaches, it may make sense to employ distributed memory programming approaches on such architectures as under this programming paradigm all data structures are local to the executing process.

3.5 Distributed Memory Programming on Shared Memory Architectures

Distributed memory programming models are typically used on distributed memory systems, i.e., systems where the individual processors do not share any common memory and therefore communicate explicitly by exchanging messages over a network. These systems and appropriate programming models will be discussed in more detail in Chapter 4. At this point, it is only important to note that applications that have been designed for distributed memory systems can usually be executed on shared memory architectures as well. For example, most applications from the high performance computing domain use the message passing interface (MPI, see Section 4.3) that follows the distributed memory programming paradigm. These applications can be run on a shared memory machine by using an MPI implementation that performs the message exchange between the individual processes over a shared memory region instead of a

communication network. A similar approach can be applied to enterprise applications that are frequently based on the client/server concept and hence on explicit message exchange.

Distributed memory programming is considered to be more complex than shared memory programming due to the fact that the programmer not only has to devise appropriate schemes for work load distribution, but also for data distribution. Hence, distributed memory parallelizations tend to be more holistic, i.e., do not only concentrate on some regions of the codeand therefore scale to a much higher degree. Clearly, similar approaches would be beneficial for shared memory implementations as well. The same parallelization strategy that can be implemented with, e.g., MPI can usually also be implemented with threads. Even though an MPI application can also be run on a shared memory system, it can be beneficial to re-implement it as a multi-threaded application: the communication overhead of a multi-threaded implementation is typically lower which allows for higher parallel efficiency. Additionally, the appropriate runtime environment that is required for the distributed memory implementation (e.g., the MPI libraries), may not be available. Contrary to that, multi-threaded applications are supported by all multi-tasking operating systems out-of the box and require no additional libraries or runtime environment and are therefore easier to install and execute for the user.

3.6 Efficient Exploitation

Independently from the programming model, the main prerequisite for achieving good performance on any parallel system evidently is efficient parallel code. This implies a suitable parallelization strategy as well as an efficient implementation of that strategy. However, a discussion of the principles of parallel programming is outside of the scope of this work. Good introductions into this field can be found in [34, 47, 119].

Yet, the new multi-core architectures show some interesting properties that are crucial for achieving good parallel performance on such systems. As these properties are not covered by the standard textbooks on parallel programming they will therefore be discussed in the following.

3.6.1 Shared Caches

The most interesting properties of multi-core architectures are due to their memory subsystem, or more specifically due to their cache hierarchy. As pointed out in Section 3.3.2, the last level cache(s) of most multi-core processors are shared caches. Those caches are – according to Intel's terminology – "smart caches". That is, there is no fixed allocation of cache lines to specific cores but they are assigned dynamically depending on memory access patterns. This is not a Intel-specific feature but is due to the functioning of caches: memory requests are referenced by their physical address in the main memory. If the cache line that corresponds to the requested address is not stored in the cache it will be fetched from main memory. Which core actually requested the cache line has no influence on this behavior. Consequently, the fraction of the shared cache that is occupied by a specific core only depends on the its own and the other cores' memory access patterns.

Shared caches can have beneficial impact on application performance on the one hand, but can also induce performance degradation on the other hand. For example, if only one sequential program is running on a dual-core processor, the whole shared cache will be occupied by the core that actually executes the program. This certainly improves the performance of the program as it has a bigger cache at its disposal compared to a processor with two separate caches of only half the size.

Of course, shared caches can also improve the performance of multi-threaded applications. Communication between two threads occurs implicitly by reading and writing shared variables. However, memory accesses are performed at cache line granularity on most microprocessor architectures as described in Section 3.3.2. Thus, exchanging a single byte between two threads running on two different processors requires that the processor executing the writing thread reads a whole cache line, modifies it, and writes it back to main memory. Afterwards, the processor that executes the reading thread can fetch the corresponding cache line from the main memory. Note that, the cache coherency protocal is responsible for writing the modified cache line back to the main memory upon a read request. This process obviously induces considerable latency which can have serious influence on the performance of the application, especially if it is performed frequently.

On multi-core processors with shared caches, communication between two threads can be significantly accelerated as data does not need to be exchanged via the slow main memory but can be exchanged via a shared cache at a much lower latency. In fact, in a study on the effects of false sharing on multi-core architectures [152] we showed that the latencies of exchanging data via a shared cache are one order of magnitude lower than the latencies induced by main memory data exchange. This could improve the performance of applications that need to frequently exchange data between threads, such as producer/consumer problems.

However, in the case of a multi-threaded application that performs frequent irregular memory accesses, shared caches can impact the application performance. On every cache miss a cache line needs to be evicted from the cache to give room for the cache line that is to be fetched from the main memory. If all cores have separate caches, only the memory accesses of the associated core can cause cache lines to be evicted. By using, e.g., cache blocking techniques the programmer can ensure that the working set of the application remains in cache as long as necessary such that all accesses to the working set can be fulfilled by the cache at high speed. However, if all cores share a cache they also compete for the cache. Consequently, the memory accesses of one core may evict the cache lines of another core which increases the pressure on the memory subsystem and contributes to the problem of the memory wall.

3.6.2 Dealing with the Memory Wall

The key figures for memory performance are latency and bandwidth, but most of the discussion of the memory wall is about latency. However, memory latency can be hidden by caches as long as the memory access patterns of the application are regular. In this case, the processor's cache controller is able to detect the pattern and will load the cache lines that are likely to be

accessed next to the cache in advance (so-called prefetching). The actual memory access can then be carried out by the cache in advance to hide latency.

Many scientific applications implement iterative numerical methods that have regular memory access patterns (e.g., iteratively loop over a large memory block) and consequently do not suffer from high memory access latencies. However, such applications are often memory bandwidth bound as their working set may not fit into the cache or their data access patterns may not exhibit temporal locality. So-called blocking strategies can be used to split the working set into smaller pieces that can be kept in cache and re-used which reduces the memory bandwidth requirements. However, if the application lacks sufficient temporal data locality, the memory bandwidth remains a limiting factor for its performance. For such applications, increasing the number of processor cores (e.g., replacing a dual-core processor by a quad-core processor) would not result in any performance gain.

The problems associated with lacking memory bandwidth are well-known from traditional multi-socket systems and have finally led to the introduction of NUMA architectures that alleviate these problems since each processor also increases the aggregate memory bandwidth on those systems. Software-wise, a viable solution for dealing with memory bandwidth shortage would be to use only a fraction of the available processor cores and hence reduce the pressure on the memory subsystem. While this certainly is not the favored approach for expensive multi-socket systems, it can increase the throughput on multi-core systems.

3.6.3 Thread Pinning

Due to the memory wall and the heterogeneous properties of the memory subsystems of modern multi-core processors binding of threads to specific cores (so-called pinning) has tremendous impact on application performance. As indicated in the previous sections, it can be beneficial for the overall performance to utilize only a fraction of the available processor cores in order to increase the memory bandwidth and the cache size that is available to each core. Within the scope of the Munich Multicore Initiative MMI[1] we assessed the impact of different pinning strategies [109] by example of the SPEC OMP benchmark suite [125] on a wide range of multi-core architectures. The experiments showed that, taking the specific characteristics of the memory subsystem into account when assigning the threads of a multi-threaded application to particular cores, can have tremendous impact on performance. In the following only, the experiments and their results on the Intel Caneland platform (see below) will be discussed as this platform allows for pinpointing the properties and potential performance pitfalls of shared memory architectures nicely. However, similar observations can also be made on other platforms [109].

The SPEC OMP benchmark suite is a collection of eleven applications that allow for assessing the performance of shared memory systems. All applications are scientific codes that are used in production environments for solving real-world problems, i.e., they are not synthetic benchmarks. The particular purpose of the individual benchmark applications is described in

[1] http://mmi.in.tum.de

Application name	Description
310.wupwise	quantum chromodynamics
312.swim	shallow water modeling
314.mgrid	multi-grid solver in 3D potential field
316.applu	parabolic/elliptic partial differential equations
318.galgel	fluid dynamics analysis of oscillatory instability
320.equake	finite element simulation of earthquake modeling
324.apsi	weather prediction
326.gafort	genetic algorithm code
328.fma3d	finite-element crash simulation
330.art	neural network simulation of adaptive resonance theory
332.ammp	computational chemistry

Table 3.1: SPEC OMP benchmark applications

Table 3.1. All benchmark applications are provided as source code. For the experiments described here the Intel Compiler Suite 9.1 was utilized for compiling the applications. There are two different levels of workload for SPEC OMP: medium and large. As the maximum number of cores used was 16, all experiments have been conducted with the medium-sized workload. The large workload is intended to be used for large scale systems of 128 and more cores.

The Intel Caneland platform consists of four 7300 series Tigerton processors [81] which implement the Core2 microarchitecture and are essentially multi-processor capable versions of the previously described Clovertown processors, i.e., they can be used in multi-socket shared memory systems. Each processor connects via a dedicated FSB (1066 MT/s, bandwidth 7.94 GB/s) to the so-called Clarksboro chipset [77]. The DDR2-667 memory modules are attached to the chipset via four channels and deliver an aggregate memory bandwidth of approximately 20 GB/s. A schematic representation of the platform is depicted in Figure 3.3. The Caneland system on which the experiments have been carried out was equipped with 2.93 GHz processors (X7350) which results in a theoretical peak performance of 187 GFLOPS. All experiments have been conducted on the Linux operating system running kernel version 2.6.23.

The most important result of the experiments is the observation that threads need to be pinned to specific cores, i.e., the operating system scheduler needs to be forced to execute a specific thread on a specific core. Without a fixed pinning, the runtimes of an application are almost impossible to predict as the thread-to-core binding chosen by the operating system scheduler typically ignores the requirements of the application as well as the properties of the memory hierarchy. Figure 3.4 shows runtimes for the SPEC OMP 314.mgrid benchmark with four threads for 50 sample runs on the Caneland platform. As additional experiments with this specific benchmark have shown (see below), it shows significant runtime variations for different thread-to-core pinnings depending on the size of available cache and the available memory bandwidth. For the "pinned" runs, an optimal thread-to-core binding has been chosen and the threads have been pinned for all runs accordingly. The runtimes of the individual sample runs therefore only vary slightly. For the "unpinned" runs, no pinning has been specified. Hence,

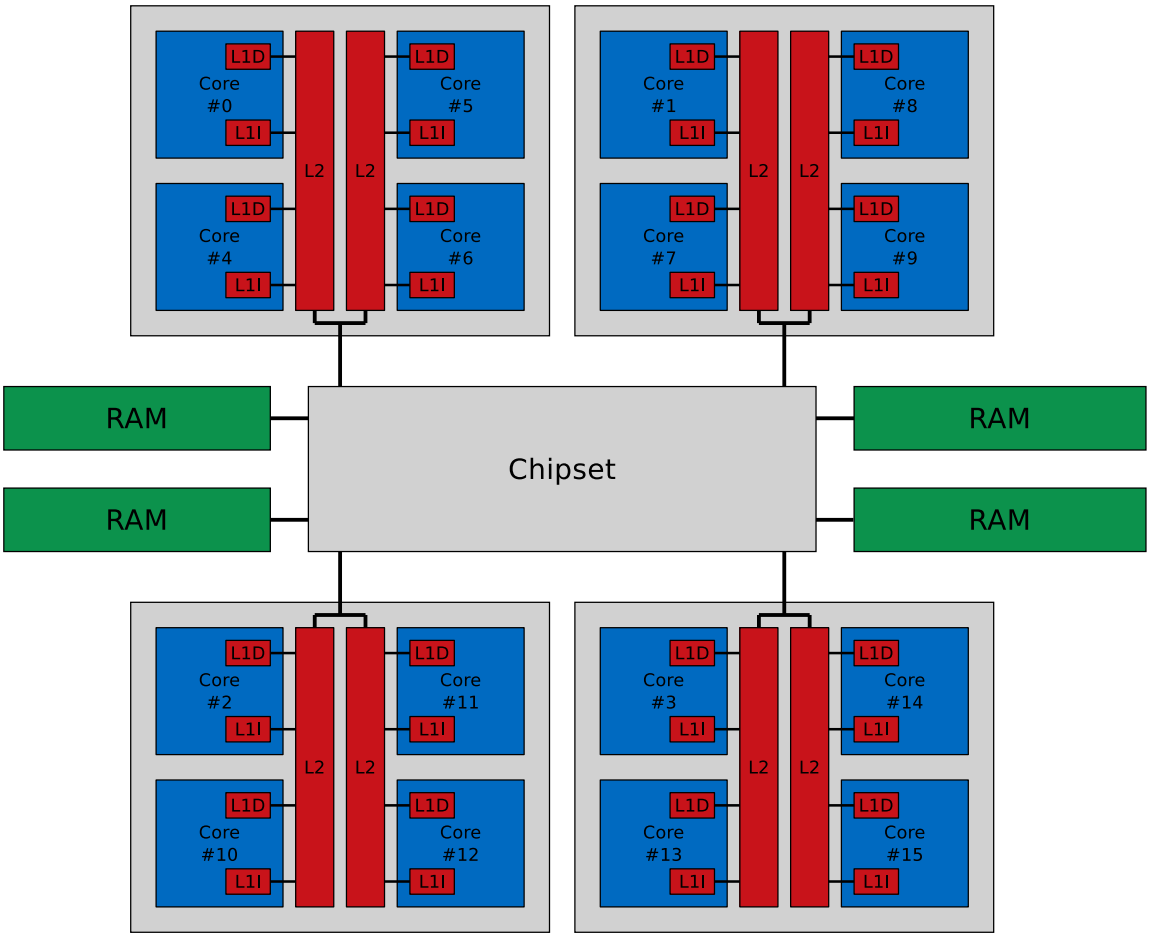

Figure 3.3: The Intel Caneland platform, consisting of four Xeon 7300 series quad-core Tigerton processors, the 7300 series Clarksboro chipset, and DDR2-667 memory modules connected via four channels.

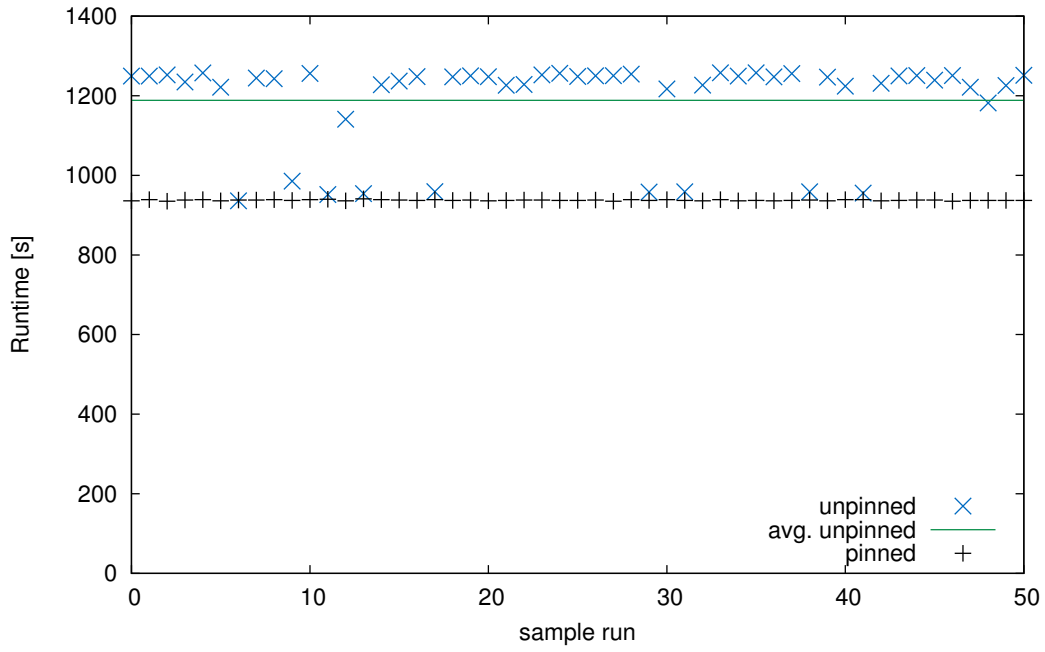

Figure 3.4: Runtimes of the 314.mgrid application from the SPEC OMP benchmark suite with four threads on the Intel Caneland platform. 50 sample runs with a fixed thread-to-core pinning, 50 runs without.

the threads have been scheduled to the individual cores dynamically by the operating system. As a consequence, individual runtimes vary significantly and are considerably higher than the runtimes of the optimal fixed pinning for the dynamically chosen thread-to-core bindings obviously being sub-optimal. On average, the unpinned runs are 27% slower than the runs with a fixed optimal pinning.

Finding the optimal pinning can be cumbersome without deep knowledge on the communication patterns of the application. Yet, for most applications the key properties that affect the performance are the cache size available to each thread and the available memory bandwidth. The particular architecture of the Caneland platform allows for "simulating" different scenarios and measuring the impact of reduced cache sizes and/or reduced memory bandwidth. Since each of the four processor sockets has a dedicated FSB, the average memory bandwidth available to each thread can be influenced by the number of threads that are being executed on the cores of a single socket. Distributing the threads evenly over all sockets delivers the highest memory bandwidth possible to each of the threads, whereas concentrating the threads on a single socket effectively reduces it. The cache size available to each thread depends on the number of threads that are being executed on cores that share a L2 cache. If only a single thread is executed, the whole L2 cache is at its disposal.

The impact of the different thread-to-core bindings can be witnessed best on runs with only two threads. Figure 3.5 shows the respective runtimes of the individual SPEC OMP benchmark applications in three different configurations:

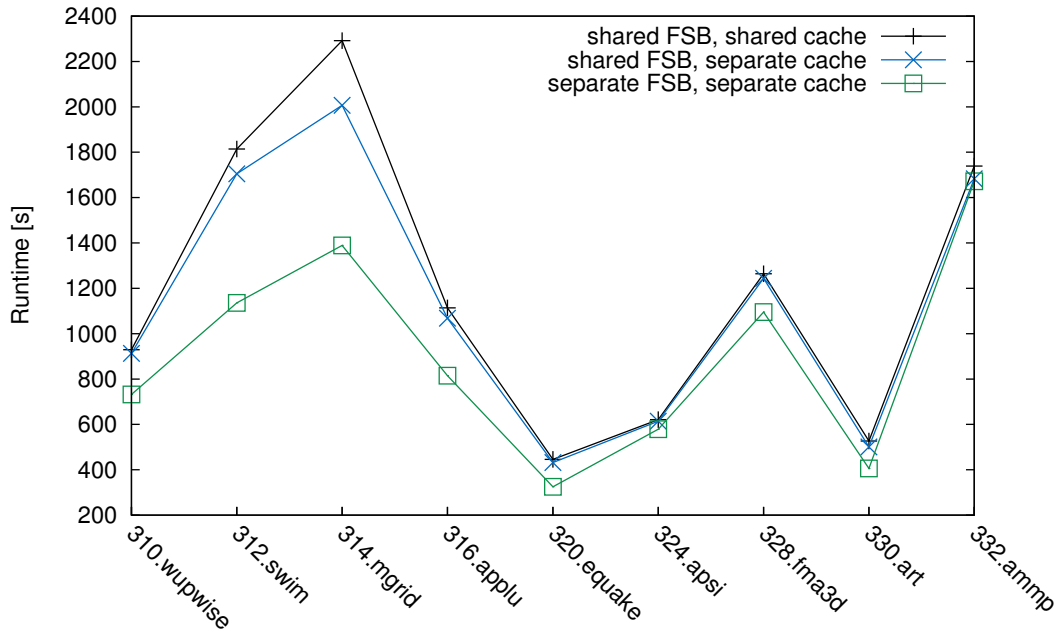

Figure 3.5: Runtimes of the SPEC OMP benchmarks with two threads in different configurations: both threads on different sockets, both threads on the same socket but with separate L2 caches, and both threads on the same socket and with shared L2 cache.

1. Both threads pinned to different sockets (denoted as separate FSB, separate cache). This configuration allows for each thread to access the main memory at the full FSB bandwidth of 7.94 GB/s, resulting in an aggregate bandwidth of 15.88 GB/s. As this is below the total available memory bandwidth of 20 GB/s, the limiting factor is the frontside bus and consequently this configuration provides the best possible bandwidth to each of the two threads.

2. Both threads pinned to cores on the same socket that do not share the L2 cache (denoted as shared FSB, separate cache). For this configuration the FSB bandwidth is shared between both threads which results in an average memory bandwidth of 3.97 GB/s per thread. Yet, each thread has a full L2 cache of 4 MB at its disposal.

3. Both threads pinned to the same socket, sharing the L2 cache (denoted as shared FSB, shared cache). With this configuration both threads share the frontside bus as well as the 4 MB L2 cache.

The influence of the memory bandwidth can be assessed by comparing the runtimes of the first configuration against the runtimes of the second configuration. The lower per-thread memory bandwidth provided by configuration #2 results in higher runtimes for all benchmark applications. Yet, the performance penalty varies over the individual benchmarks: while the decreased memory bandwidth causes an increase in runtime of 50% for 312.swim and 44% for

314.mgrid, the impact is significantly lower for 324.apsi (6%) and almost negligible for 332.ammp (0.65%).

The impact of shared caches can be estimated by comparing configuration #2 against configuration #3. The experimental results show that all benchmarks run faster when executed on cores with separate caches. Yet, the performance decrease induced by shared caches is not as prevalent as the impact of the memory bandwidth. It varies between 1.2% and 14.2%. Again, 312.swim and 314.mgrid show the biggest runtime difference of all benchmarks (6.4% and 14.2% respectively). For all other benchmarks the difference in runtime is below 5%. Note, that the increase in runtime due to sharing of caches adds to the increase caused by the reduced per-thread memory bandwidth. If compared to configuration #1, the total runtime increases up to 65%.

Interestingly, no application benefits from shared caches. This is probably due to the fact that the SPEC OMP benchmark suite has already been released in 2001 when no multi-core processors with shared caches were available. Consequently no measures have been taken to exploit the advantages of shared caches.

The importance of thread pinning for higher thread counts can be derived from the runtimes with one, two, four, eight, and sixteen threads that are depicted in Figures 3.6 and 3.7. For the runs with two, four, and eight threads the times with the optimal pinning (all threads distributed evenly over the sockets, no sharing of caches) and the most unfavorable pinning (all threads concentrated on as few sockets as possible, shared caches) are shown.

Most applications of the benchmark suite show almost linear scaling behavior with two threads – if the threads are executed on suitable cores. If an unfavorable pinning is chosen, the applications benefit only slightly from the second core and consequently the runtimes of a bad two-core pining are only marginally lower than the reference runtimes on a single core. The impact of a bad thread pinning increases with the total number of threads: on average, the runtimes of the worst four-core pinning are over 74% higher than the optimal pinning. For 312.swim and 314.mgrid the difference in runtime exceeds 110%. For six out of nine benchmark applications the runtimes with a unfavorable four-core pinning are higher than the runtimes with the optimal two-core pinning. Similar effects can be observed for the same six benchmarks when comparing the optimal four-core pinnings with bad eight-core pinnings (see Figure 3.7). Additionally, only three benchmarks (324.apsi, 328.fma3d, and 332.ammp) benefit from doubling the number of cores from eight to sixteen. Four benchmarks (310.wupwise, 312.swim, 316.applu, 330.art) show almost identical runtimes regardless of whether they are executed on eight or sixteen threads which indicates that for these applications the available memory bandwidth is saturated by only eight threads. For 314.mgrid and 320.equake the performance even degrades when executed on sixteen cores instead of eight cores with an optimal pinning as the runtimes increase by more than 13%. Obviously, the sharing of caches in case of the sixteen threads runs degrades the performance.

The observations can be summarized as follows:

- **Pinning is important for obtaining reproducible runtimes.** Otherwise the operating system scheduler decides on the thread placement which might be sub-optimal.

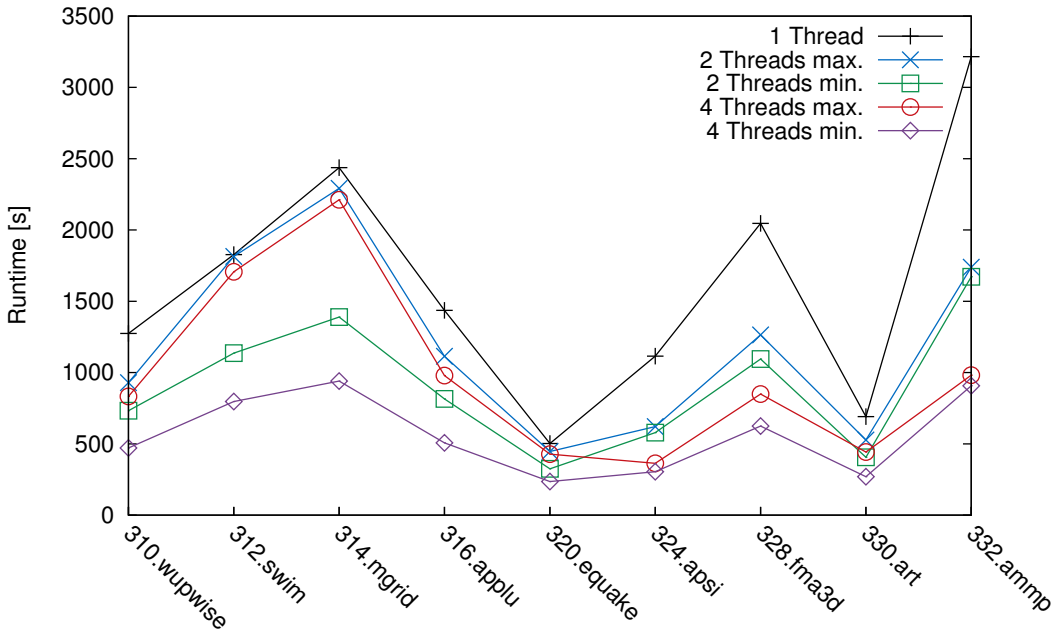

Figure 3.6: Runtimes of the SPEC OMP benchmarks with one, two, and four threads. The runs with two and four threads have been conducted twice, once with the optimal pinning and once with the most unfavorable pinning.

Additionally, the threads might migrate from one core to another which results in cache trashing and consequently in decreased performance.

- **Per-thread memory bandwidth is critical for performance.** On multi-core systems, applications often saturate the available memory bandwidth with only one thread per socket. Additional threads might even decrease application performance.

- **Shared caches allow for higher flexibility.** The potential benefit of shared caches (fast data exchange between threads) needs to be exploited explicitly by the developer. If the application does not account for shared caches, they can limit the performance.

Especially the last two observations suggest to utilize only a fraction of the available processor cores for some applications. Since this increases the memory bandwidth as well as the cache size available to each thread it can have beneficial impact on application performance. Additionally, as modern microprocessors support so-called deep sleep states for un-utilized cores [70], this approach also allows for saving energy.

3.7 An Outlook on Automated Thread Pinning: `autopin`

Most computer users have no knowledge of neither the performance properties of the computers they are using nor of the requirements and characteristics of the applications they are exe-

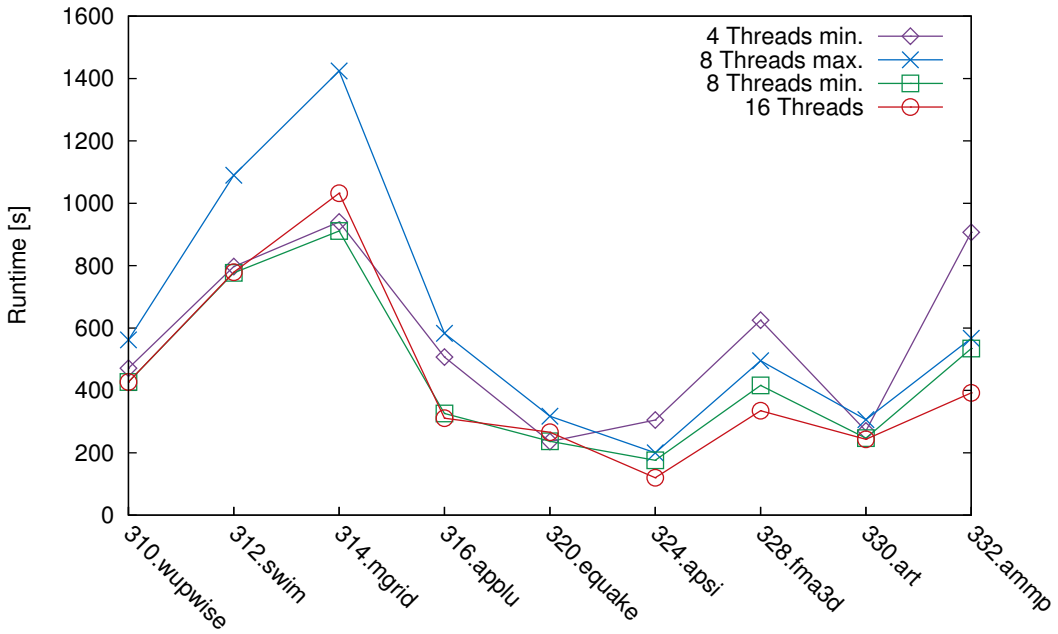

Figure 3.7: Runtimes of the SPEC OMP benchmarks with four, eight and sixteen threads. For the run with four threads the optimal pinning was used, for the runs with eight threads the runs have been conducted with the optimal as well as with the possibly worst pinning.

cuting. However, the previously described experiments with the SPEC OMP benchmark suite show that such knowledge is necessary for proper thread pinning and efficient exploitation of multi-core systems. Though the assignment of threads to processors should be done by the operating system scheduler, the assignments chosen by most schedulers yield only poor performance on multi-core systems for not taking the system's as well as the application's properties into account. Hence, an additional tool is required that automatically detects the properties of the hardware, gathers the characteristics of the application, and pins the application threads accordingly. As a follow-up to the performance study on the SPEC OMP benchmark suite we developed such an automatic tool and described it in [86] and [109]: `autopin`. The implementation details of the tool as well as the experimental results are discussed in detail in [109]. Here, only the conceptional ideas of the `autopin` tool will be described.

The main idea of `autopin` is to retrieve all necessary information about the application characteristics from CPU hardware performance counters at runtime. Hardware performance counters (often also called performance monitoring unit PMU) are built into every processor on the market today. In fact, similar counters have always been present in processors to allow for internal correctness checks after production. Only recently, processor manufacturers started to document the counters and make them available to the user (for example, see the manuals for Intel [76] or AMD [5] processors). Typically, there are 2 or 4 counters available for a huge number of event types related to the processor pipeline, the cache subsystem, and the bus

interface. They can either be used to read exact counts, or to derive statistical measurements. However, the semantics of events can be difficult to interpret, and often, detailed documentation is not available. Additionally, the amount of different events that can be measured varies from processor type to processor type and from vendor to vendor. So far no standard has been established that determines which performance counters have to be present in a processor and how they can be accessed. Hence, an additional software layer is required that provides generic access to performance counters. Probably the best known tool for accessing performance counters is the *P*erformance *API* (`PAPI`) [17] which provides a generic performance counter API for multiple platforms and operating systems. Additionally, there exist various tools for raw read access to performance counters like `Perfctr` [117] and `perfmon2` [36] to name but a few. `perfmon2` was originally developed by Hewlett Packard exclusively for the Linux Itanium architecture, but now also provides support for latest Intel and AMD processors. The proof-of-concept implementation of `autopin` relies on `perfmon2` for accessing the performance counters.

If related to a specific time interval or program region, many events that can be measured by the PMU allow for deriving metrics that can be used to assess the performance of an application. An important metric for any kind of application certainly is the number of instructions executed in a given time interval which is typically referred to as MIPS (million instructions per second). Under the assumption that the number of instructions is fixed for a certain problem, a high MIPS number indicates that the problem has been solved in short time whereas a low MIPS number indicates poor performance. As the maximum number of instructions a processor can execute per second is known, relating the measured MIPS to the theoretical maximum additionally allows for computing the efficiency of the execution. For floating point intensive applications a more interesting metric is FLOPS, the number of floating point operations executed per second. Both metrics can be easily obtained by counting the number of executed (floating point) instructions in a given time interval and dividing them by the length of the interval. More elaborate metrics can be derived by combining several performance events. For example, the efficiency of cache usage – the cache hit rate – can be measured by relating the number of cache hit events to the number of memory access events. For memory bound applications a low cache hit rate has significant impact on the performance and therefore is a suitable metric for its assessment.

Given a suitable performance metric and a set of reasonable thread-to-core pinnings, the information provided by the PMU can be used for choosing the optimal pinning of an application during runtime. In the proof-of-concept implementation of `autopin` each pinning from the set is probed for t seconds using the following algorithm:

1. Let the program initialize for i seconds.
2. Read the current value pc_1 of the performance counter for each thread.
3. Run the program for t seconds.
4. Read the current value pc_2 of the performance counter for each thread.

5. Calculate the performance rate $r_j = (pc_2 - pc_1)/t$ for each thread j and the average performance rate r_{avg} over all threads.

6. If further pinnings are left for probing, re-pin the threads according to the next pinning in the list, let the program "warm up" for w seconds, and return to 2.

The initialization step in 1. is required to avoid measuring potential sequential phases in the initial stage of the program [48]. The "warm up" time after each re-pinning is needed for the actual rescheduling of the threads and to refill the cache. All parameters t, i, and w can be specified in the command line. If the parameters have not been specified, the default values will be used which have proven to be suitable in previous experiments: $t = 30s$, $w = t/2$, and $i = w$. If no specific performance counter event is specified by the user, the number of retired instructions will be counted by default. However, any event that `perfmon2` provides for the given hardware platform may be used instead. The specific average performance rate r_{avg} of each pinning is written to the console. After all pinnings have been probed, `autopin` displays the pinning that achieved the highest performance rate and re-pins the threads accordingly. The program then continues execution with this optimal pinning which will not be changed until the program terminates. Additionally, every t seconds the current performance rate is calculated and written to the console. Note that, this behavior could be inappropriate for programs that have strongly varying execution phases with different properties. For such applications it might be necessary to re-pin the threads every time they enter a new phase. As different program phases might be difficult to detect automatically, this would probably require user support, e.g., by means of source code instrumentation, which would be contrary to `autopin`'s automation approach. In addition, frequent re-pinning of threads results in cache trashing which typically degrades application performance. Therefore we explicitly decided not to include this feature, although it could be easily added to `autopin`.

Obviously the number of possible pinnings grows exponentially with the total number of cores in the system. However, due to symmetries of the hardware architectures most combinations are redundant and therefore do not need to be probed. In general, the pinnings of interest are those that emphasize specific characteristics of the architecture: maximized memory bandwidth (distributing the threads equally over all sockets) or sharing of caches (concentrating the threads on as few sockets as possible). In the proof-of-concept implementation of `autopin` the thread-to-core pinnings to be probed have to be specified by the user. However, future versions should determine these pinnings automatically in order to also allow less experienced users to benefit from `autopin`. In fact, the necessary information about the memory hierarchies can be easily obtained from the operating system and from configuration data that can be read from the CPU [149].

Thread migration on NUMA systems poses additional challenges: as accesses to non-local memory can decrease performance, not only the thread itself has to be migrated, but also its referenced memory pages. On operating systems that support a next touch memory policy, a memory page will always be stored in the local memory of the last processor that has written to the page (within certain thresholds to avoid frequent page migration). Hence, memory pages

will be migrated automatically after thread re-pinning. However, Linux implements a first touch memory policy. That is, memory pages are fixed to the local memory of the processor that has written to it first. Although the current stable Linux kernel allows for page migration, it has to be performed manually for each page. Yet, a patch for the development branch of the kernel is available that provides the required auto-migration functionality [128]. On NUMA systems where this patch has been applied, `autopin` triggers the automatic page migration upon thread migration.

The effectiveness of the `autopin` approach has been assessed using the SPEC OMP benchmark suite on UMA as well as on NUMA architectures. A detailed description of hardware platforms, the experimental setup, and the results can be found in [86]. At this point only a comprehensive summary of the results will be given: in nearly all experiments the pinnings suggested by `autopin` yielded runtimes that did not exceed the optimal pinning's runtime by more than one percent. Only in rare cases a pinning with a total runtime of less than 5% above the optimal pinning's runtime was found. Given the fact that a bad pinning can cause an increase in application runtime of more than 100% (see section 3.6.3), the use of `autopin` can be considered to have beneficial impact on the runtime in spite of the slight overhead that might be induced.

Chapter 4

Distributed Memory Systems

This chapter describes the basic concepts of distributed memory systems and gives an overview of the Message Passing Interface which is the prevalent programming paradigm on those systems. Additionally, it describes novel concepts for detecting performance bottlenecks in large-scale MPI applications based solely on summary information as well as a tangible implementation of these concepts in the automatic performance analysis tool `Periscope`.

4.1 Introduction

Besides the fact that the shared memory programming paradigm is comparatively easy to understand and to implement, one of the main benefits of shared memory multiprocessors is their cost-efficiency. The acquisition costs as well as the operating costs of a two socket machine are only marginally higher than the costs of a single socket machine. Yet, it provides twice the performance. Although this argument typically also holds for four socket machines, the costs for bigger shared memory systems grow disproportionately. Systems with eight and more sockets only pay off, if additional benefit can be obtained from the shared memory architecture itself. For example if applications are to be executed that do not allow for a distributed memory parallelization approach or if such an approach would result in poor performance compared to a shared memory parallelization. If a shared memory architecture is not necessarily required, distributed memory systems typically provide a much better cost/performance ratio for large-scale machines.

This is also reflected by the Top500 supercomputer list [3] as can be seen from Figures 4.1 and 4.2. Both in terms of number of installed systems as well as in terms of the performance share, shared memory multiprocessors (denominated as *SMP* in the figures) have constantly been losing market share. Since 2003 no shared memory architecture has been ranked among the 500 fastest supercomputers in the world. The majority of systems belongs to the class of (commodity) *Clusters*, *Constellations*, and *MPPs* (Massively Parallel Processors). The term Constellation refers to a cluster of SMP machines. Yet, the same description also applies to most of current commodity clusters that are typically – due to the previously mentioned cost efficiency – built of small shared memory machines with two or four sockets. Both classes can be

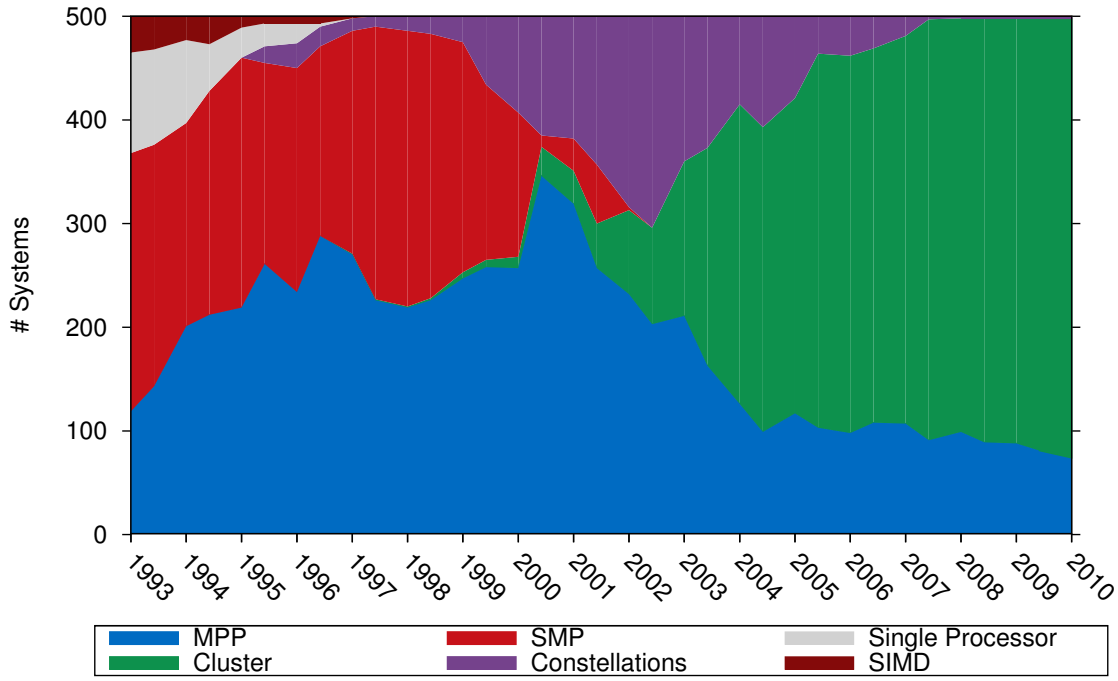

Figure 4.1: Share of different system architectures in the Top500 supercomputer list in terms of number of systems.

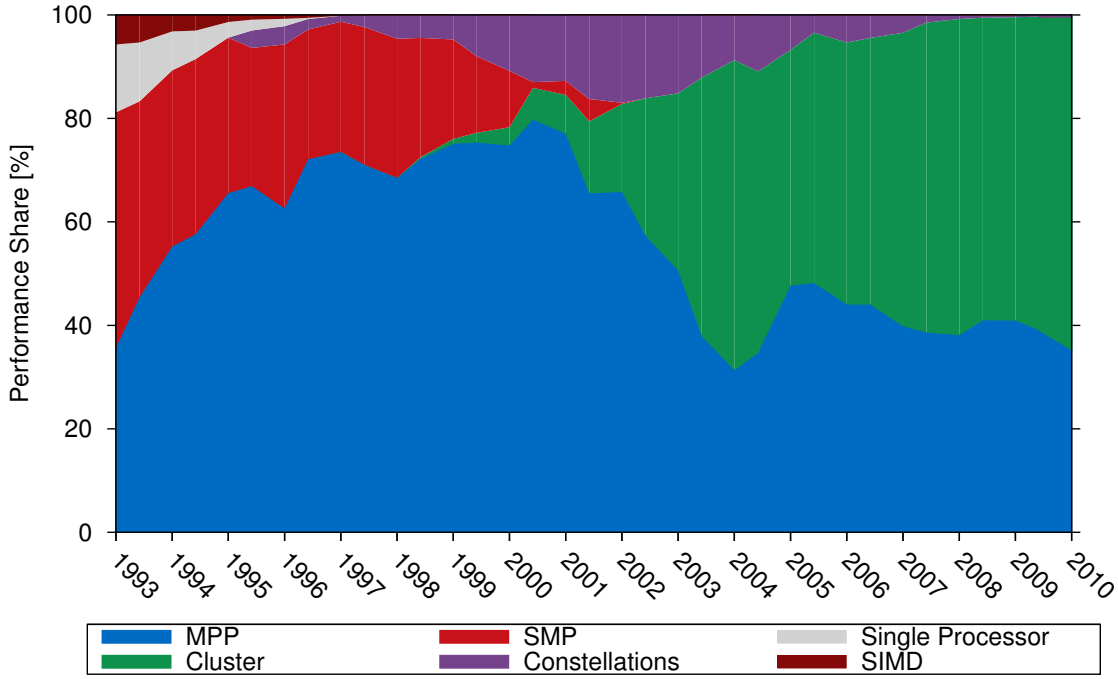

Figure 4.2: Share of different system architectures in the Top500 supercomputer list in terms of performance share.

distinguished by the level of intra-node parallelism: if the number of processors within a node outnumbers the total number of nodes, i.e., the system focuses on intra-node shared-memory parallelism, the system belongs to the Constellation class. Otherwise it belongs to the class of Clusters. Either way, since no memory is shared between individual nodes both classes are distributed memory architectures. The term MPP in the context of the Top500 list is not clearly defined and encompasses shared memory as well as distributed memory architectures. However, as of June 2010 the majority of MPP systems in the Top500 list are IBM BlueGene and Cray XT3, XT4, and XT5 systems, all of which are distributed memory architectures.

4.2 Characteristics

Though the Top500 terminology distinguishes between Clusters, Constellations, and MPPs, most of the distributed memory systems belonging to these classes implement very similar architectures from an abstract point of view: the building primitive of these systems is a single node that encompasses one or more processor sockets and local memory. Each processor socket may be equipped with a single- or multi-core processor. In case of multi-socket nodes the memory is typically shared by all processors (SMP nodes) and may be implemented with UMA as well as with NUMA. Each node has its own address space, i.e., data stored in the local memory of one node can not be accessed transparently by a processor in another node but needs to be transferred explicitly from remote memory to the local memory. Between two nodes, communication is performed over a high performance network whereas it is performed via shared memory within a node.

The prevalent programming paradigm for such distributed memory systems is message-passing, i.e., processes communicate with each other by explicitly exchanging messages. Due to this programming paradigm, distributed memory architectures are often also called *message-passing* architectures. Although numerous message-passing frameworks and implementations like CORBA [150], Java RMI [142], DCOM [99], and SOAP [156] exist, most applications in the HPC domain utilize the *Message-Passing Interface (MPI)* which will be described in more detail in the next section.

Programming under the message-passing model is substantially different from sequential or shared memory programming. The programmer needs not only to devise an approach for distributing the computations over a set of processors, but also an approach for distributing the data required for the computation. In a shared memory environment, a processor is able to fetch required data from remote memory transparently. Under the message-passing paradigm, however, the required data have to be transferred explicitly and – most importantly – in time. If the data do not arrive in time, the receiving processor is forced to wait for it and waste CPU cycles idling. Transforming a parallel application that has been designed for a shared memory environment into a message-passing application may be difficult or even impossible as both programming paradigms are diametrically opposed to each other. For example, the limited amount of main memory of a single node may prevent a distributed memory approach for some problems as the working set of each process would exceed the available memory. However, just like

a distributed memory application can be executed on a shared memory architecture, a shared memory application can also be executed on a distributed memory system by introducing an additional abstraction layer: *Distributed Shared Memory (DSM)* allows for establishing a single virtual address space over all nodes of a distributed memory system and hence facilitates the execution of applications for shared memory systems without any code modifications. A DSM system may be implemented in hardware as well as in software, but both approaches have specific drawbacks. Hardware DSM systems are typically very costly to implement as additional hardware is required not only for transparently accessing remote memory but also to ensure cache coherence. Software-based DSM implementations can provide the same functionality at significantly lower cost, but due to increased memory access latencies caused by additionally required software layers their performance is typically inferior to hardware-based implementations. Due to these drawbacks, DSM is rarely used for high performance computing. If, however, porting an existing application from shared memory communication to message-passing would not pay off due to the required development effort, software-based DSM may be a viable and cost-effective solution.

As communication between the individual nodes of a distributed memory system is performed over a network, its performance is crucial for overall system performance. In general, the performance of a network can be described by two metrics: bandwidth and latency. However, the influence of both metrics on the performance of an application depends on the application's communication patterns and the size of the messages being exchanged. The transmission time t of a message, i.e., the time that is required to transfer the message from the sender to the receiver, is described by the following function:

$$t = l + \frac{s}{b} \tag{4.1}$$

with l being the network latency, s the message size, and b the network bandwidth. Note, that the parameters of this formula may be further decomposed. For example, the latency can be split into hardware induced latency and software induced latency, and in case of the bandwidth one needs to distinguish between the (theoretical) maximum throughput that can be delivered by the hardware and the so-called goodput, which is the bandwidth that is actually available to the application. However, from an application programmer's point of view, only the overall (i.e., combined hardware and software) latency as well as the total bandwidth available at the application level is of interest. From Equation 4.1 one can easily conclude that the transmission time of a short message (a couple of bytes) mainly is a function of the latency, whereas the transmission time of large messages (megabytes) mainly depends on the bandwidth. Consequently, short latencies are important to achieve good performance with an application that frequently transfers short messages, whereas an application that only exchanges large data chunks benefits from high bandwidth. With Gigabit Ethernet now being the de-facto standard even for small local area networks, providing sufficient network bandwidth typically is not an issue when building distributed memory systems. Without spending additional money on the interconnect, a custom Linux cluster that only utilizes the on-board Gigabit Ethernet device

provides enough network bandwidth for most applications. This is also reflected by the Top500 supercomputer list: as of June 2010, 242 out of 500 systems (48.4%) utilize a Gigabit Ethernet interconnect. However, those systems only account for 30.7% of aggregate peak performance. This is probably due to the fact that many scientific applications require short latencies for achieving good performance. With latencies in the range of $10 - 20\mu s$ [26], Gigabit Ethernet is not suited for such applications. Hence, larger installations typically utilize low-latency technologies like Infiniband, Myrinet, or even proprietary interconnects that allow for latencies of $1 - 4\mu s$ at the MPI level. In fact, 207 systems (41.4%) that account for 46.6% of aggregate peak performance of the June 2010 Top500 list utilize Infiniband, and 30 systems (6.0%) with an aggregate peak performance share of 19% – most of them being IBM BlueGene and Cray XT3, XT4, and XT5 installations – have proprietary interconnects. The IBM Bluegene even features three different proprietary networks for different communication patterns: a 3D torus for peer-to-peer communication, a global collective network for efficient collective operations, and a global interrupt network for fast barrier operations. Yet, low-latency interconnects come at a significantly higher price than Ethernet interconnects and may account for up to 30% of single node cost whereas Gigabit Ethernet almost comes for free due to economies of scale effects.

4.3 The Message-Passing Interface MPI

MPI is a specification for a message-passing API developed and published by the MPI-Forum, a loosely-coupled consortium of parallel computer vendors, users, and software developers. It is the de-facto standard for developing and executing applications under the message-passing paradigm on distributed memory as well as on shared memory parallel computers. The MPI-Forum itself only publishes the specification of a message-passing library interface, actual implementations of the library are provided by third parties, which are typically members of the MPI-Forum. Most vendors of parallel computer systems like IBM, SGI, and Cray provide their own MPI implementations that are tuned for their particular computer architecture. Additionally, portable open-source implementations like LAM/MPI [21], OpenMPI [58], and MPICH [62] are available for almost any hardware platform and are therefore frequently used for custom compute clusters.

The MPI standard can be divided into two parts: MPI-1 and MPI-2. MPI-1 comprises the original functionality of version 1.0 of the MPI standard that essentially covers point-to-point and collective communication and entails a static runtime environment. Subsequent versions of MPI-1 added no additional functionality but covered errata and refinements of the standard. The initial version 1.0 of the standard was released in May 1994, the most recent version 1.3 in May 2008 [96]. New functionality has been added with the MPI-2 branch and the release of version 2.0 of the MPI standard in November 2003. MPI-2 also covers remote memory operations (RMA), parallel I/O, and dynamic process management. The most recent version 2.2 has been released in September 2009 [97]. MPI-2 can be considered as a superset of MPI-1 as it includes all of its functionality. All versions of the MPI standard are forward-compatible,

i.e., a valid MPI-1.1 program is a valid MPI-1.3 program and a valid MPI-2.2 program.

Besides MPI I/O that covers parallel file operations, the additional features of MPI-2 are hardly used by the HPC community [148]. This is probably due to the fact that most scientific applications have a long history and had been implemented using MPI-1 years before MPI-2 was specified. Additionally, the message-passing paradigm, which is completely covered by MPI-1 is minimalistic, straight-forward, and easy to understand. In contrast to that, many innovations in MPI-2, such as one-sided communication, that allows for remote memory accesses, are complex, cumbersome, and error-prone. The dynamic process management facilities of MPI-2 are typically not supported on most supercomputer installations as they do not harmonize well with most batch queuing systems. Therefore, the remainder of this chapter will only cover the widely used MPI-1 standard. A detailed description of MPI-2 can be found in [61] and a comprehensive introduction on programming with MPI-2 is provided in [63].

4.3.1 MPI-1 Functionality

In the MPI terminology, all payload data that is transferred between a set of processes is a message. A message can be as simple as a single character or byte but can also be a complex user-defined data structure. Messages to be sent are passed to the respective MPI function by reference, i.e., the address of the buffer that contains the actual message is passed. Additionally, the length of the message needs to be provided as well as the datatype of the message. For the standard C/C++ and Fortran datatypes there exist corresponding MPI datatypes like `MPI_CHAR`, `MPI_INT`, and `MPI_DOUBLE`. For complex datatypes like C structures, a matching MPI structure needs to be defined using the `MPI_Type_*` functions. Receivers need to provide a buffer that is large enough for storing the message which is the programmer's responsibility.

All communication is based on so-called *communicators* that establish a communication context between a set of processes. Upon initialization of MPI, each process is assigned to the so-called *world-communicator*, `MPI_COMM_WORLD`. Custom communicators can be created at runtime, e.g., for splitting processes into functional groups or for fitting the communicator to the network topology. Within a communicator, each process has a unique identifier (called *rank*) and all exchanged messages are ordered. That is, messages arrive in the same order at the receiver in which they have been transmitted by the sender. The messages sent within one communicator do not interfere with the messages of any other communicator as both communicators are logically independent communication channels.

The communication operations in MPI-1 can be divided into two groups: point-to-point and collective operations. Point-to-point operations cover the message exchange between two single processes, whereas collective operations handle the message exchange between a group of processes. Communication partners are specified either by a communicator only (group of processes) or by a communicator combined with a process rank (single process). Communication between processes is explicit, i.e., each communication partner needs to call the respective send or receive function. Note, that there is no distinction between send or receive functions for collective operations – all processes call the very same MPI function.

4.3.1.1 Point-to-Point Operations

The point-to-point communication pattern mainly comprises two functions: send and receive, both of which are available as blocking (`MPI_Send()` and `MPI_Recv()`) as well as non-blocking versions (`MPI_Isend()` and `MPI_Irecv()`). From the sender's perspective there is only little difference between both types. In both cases the send function may return immediately, no matter whether the message has already been transferred to the receiver or not. The main difference is the fact, that a blocking send guarantees that it is safe to modify the buffer containing the message after the send function has returned. In case of a non-blocking send, the programmer needs to check explicitly whether the message has already been transferred. However, the semantics of non-blocking and blocking receive operations differ fundamentally: the blocking receive function will return only after the message has been received and copied to the message buffer provided by the programmer. The non-blocking receive function on the other hand will return immediately. Its single purpose is to provide a buffer for the message to be received. Whether the message has actually been received needs to be ascertained explicitly by the programmer with `MPI_Wait()` afterwards.

4.3.1.2 Collective Operations

All collective MPI-1 functions are blocking. That is, a sender's function call will return immediately after the message has been copied to a send buffer of the runtime environment, but a receiver's function call will return only after the message has been received and copied to the user-provided receive buffer. They can be divided into three classes: 1-to-N, N-to-1, and N-to-N.

1-to-N The 1-to-N class comprises functions with a single sender and a group of receivers like `MPI_Broadcast()` or `MPI_Scatter()`. The broadcast operation allows for sending a single message to all processes belonging to the specified communicator. The scatter function allows for distributing a vector to a group of processes, i.e., each process receives only a fraction of the vector. The default distribution scheme of the vector is round-robin but a custom scheme may be specified by the programmer (`MPI_Scatterv()`).

N-to-1 The counterpart to scatter in the N-to-1 class of functions is `MPI_Gather()`. The data received from the group of sending processes are re-assembled into a single vector at the receiving process. Analogously to `MPI_Scatterv()` there also exists a `MPI_Gatherv()` function that allows for specifying a custom vector distribution scheme. Another important N-to-1 function is `MPI_Reduce()`. It is similar to gather as it collects data vectors from the senders at the receiver. However, the vectors are not re-assembled into a larger vector, but a reduce operation is performed on the received vectors. In other words, the corresponding vector entries of each sender are combined by the reduce operation and the result is written to a vector of the same length at the receiver. The programmer can either use one of the pre-defined reduce operations like, e.g., min, max, add, and multiply, or define a custom reduction.

N-to-N In the class of N-to-N operations, each process of the specified communicator is a sender as well as a receiver. The most intuitive example of such an operation is the `MPI_Alltoall()` function: each process of the communicator sends a message to all other processes and receives a message from them in return. Hence, each process has to pass a receive buffer to the function large enough to store $N - 1$ messages. There also exist N-to-N counterparts of the gather and reduce operations (`MPI_Allgather()` and `MPI_AllReduce()`). Their functionality is similar to the N-to-1 versions, but the result of the gather or reduce operation is assembled not only at a single receiving process but at all processes. A special case of an N-to-N function is `MPI_Barrier()` as it exchanges no data between the individual processes but only synchronizes them. That is, each process that calls the barrier function will return from the function call only after it has been called by all other processes. It is important to note that *all* N-to-N functions are in fact synchronous. Therefore, they should be used rarely, because for instance, a single process that enters the respective function later than others forces all processes to idle and hence wastes compute cycles.

4.3.2 The MPI Profiling Interface

Besides the core communication facilities, the MPI standard also defines a profiling interface for MPI. Profiling the communication behavior of a parallel application is essential for pinpointing performance bottlenecks that affect the scalability and consequently the overall performance of the application (see Section 4.4). The main idea behind MPI's profiling interface is to intercept all MPI function calls by wrapper functions that acquire the desired profiling information and call the original MPI function. Every function of the MPI library exports not only its common name, but also the very same function name prepended by a 'P', e.g., the `MPI_Send()` function may also be called as `PMPI_Send()`. On systems that support weak external symbols, the MPI library defines the `MPI_*` functions as weak, such that they can easily be overloaded by another function with the same name. For example, to profile the `MPI_Send()` function, one needs to write a wrapper function called `MPI_Send()` that collects the profiling information and calls the corresponding `PMPI_Send()` function for the actual MPI functionality. The wrapper functions need to be compiled into a shared library that has to be linked with the application to be profiled. It is not necessary to provide wrappers for all MPI functions: for function names which are not exported by the profiling library, the weak symbol of the original MPI library still holds. Note however, that the MPI standard defines bindings for both the C/C++ programming language as well as Fortran. As a consequence, a profiling library must not only provide a single wrapper for each MPI function to be profiled but two: one for the C/C++ binding and one for Fortran (e.g. `MPI_Send()` and `mpi_send_()`).

One of the main advantages of the MPI profiling interface is that it is transparent to the application: no code instrumentation or any other modifications are necessary, and on most systems the application does not even need to be recompiled. Additionally, as the interface is defined by the MPI standard itself, any profiling tool that utilizes this interface is independent from the actual MPI implementation.

4.4 An Outlook on Automated Performance Analysis: `Periscope`

During my work with distributed memory architectures, I also developed novel concepts for detecting performance bottlenecks in MPI applications automatically. They are based solely on summary information and therefore allow for analyzing large-scale applications with only little overhead. I integrated those concepts into the `Periscope` tool for automatic performance analysis that is being developed at the Lehrstuhl für Rechnertechnik und Rechnerorganisation. This section outlines the basics of automated performance analysis, introduces the `Periscope` tool, and describes the concepts and implementations for detecting MPI performance bottlenecks.

4.4.1 Automated Performance Analysis

The computational power of supercomputers grows at a significantly higher rate than what would be expected according to Moore's law. This performance increase is achieved by an increased level of parallelism – in terms of cores per socket as well as in terms of total number of sockets in the system. Five years ago, the majority of systems in the Top500 supercomputer list (395 systems, 79%) consisted of ≤ 1024 cores and only 16 systems of more than 4096 cores. As of June 2010, the majority of systems (387 systems, 77.4%) comprises more than 4096 processor cores, with five systems encompassing more than 128,000 cores. The number one system even comprises 224,162 processor cores. This trend is about to continue with more and more systems entering the Petascale league.

Developing applications for such massively parallel architectures is a challenging task and good scalability may be hard to achieve. However, due to the enormous costs of such machines – with respect to both acquisition as well as operation – a high level of efficiency is required. Yet, optimizing an application for best performance in the design phase is almost impossible due to the complex architecture of such machines. Donald Knuth even points out that "premature optimization is the root of all evil" [87] as programmers often tend to focus on performance too early in the development process. Hence, performance optimization is usually conducted after the main development of an application has been finished and its features have been tested for functional correctness. Additionally, most scientific codes have a long history and have usually been ported repeatedly from one hardware architecture to another over time. The source code of those applications is typically sufficiently portable and therefore compiles with only little modifications on a new platform. Yet, the time-consuming task of the porting process is optimizing the code for high performance on the new hardware architecture. Hence, performance analysis is a crucial task during the development and adaption of applications for high-performance computers.

Performance optimization is typically not straight-forward but an iterative process with alternating phases of performance analysis and application tuning. This development process if often referred to as the *measure-analyze-modify* cycle [8, 101], as the performance analysis step actually consists of two steps: measuring performance-critical properties of an application

and analyzing the measured data. While the measuring step is typically predetermined by the actual properties to be measured, there are multiple approaches for handling and analyzing the measured data.

Most traditional performance analysis tools for parallel systems are so-called trace-based tools. That is, they collect information on performance critical events during the execution of an application and write these event-traces to an output file. After the application has finished, the traces are visualized by the tool and interpreted by the developer. However, the increasing level of parallelism required to exploit the full potential of current supercomputers also increases the complexity of the applications. Manually analyzing trace files of such large-scale applications and identifying performance bottlenecks therefore is challenging if not even impossible. Yet, as scaling down the application (i.e., executing it with a lower processor count) also alters its performance characteristics, actual program runs with real data and the desired number of processors have to be analyzed to detect performance problems and their causes. Hence, tools are required that automatically collect, filter, and analyze the performance data of a given application and pinpoint the code regions that impact performance.

Automated performance analysis has been a research topic for more than 15 years now. Over the time many tools have been developed, the most notable ones being Paradyn [100], TAU [129], Vampir [19], KOJAK [155], and SCALASCA [50]. The automation of the analysis process by these tools typically aims at detecting pre-defined properties that are known to cause performance bottlenecks in applications. For MPI applications most performance critical problems are related to load imbalance that causes a subset of the processes to wait for communication partners that need more time to process their work share and thus call the respective MPI function too late. This waiting time can be measured by utilizing the MPI profiling interface which also allows for identifying the respective MPI function calls that caused the delay. A list of frequently observed performance bottlenecks and load imbalance problems in MPI applications has been assembled by the APART[1] working group [39].

Most performance analysis tools are trace-based, i.e., they perform the analysis step after the application to be analyzed has terminated (post-mortem). However, collecting information about all performance-relevant events of a large-scale parallel application can easily produce a huge amount of data. Without further filtering, this mass of information would not only overwhelm the user but may also be a strain for the machine as those amounts of data have to be stored in main memory or written to hard disk during the application's runtime. `Periscope` takes a different approach to automated performance analysis. Its analysis is solely based on summary information: instead of keeping record of every single performance-critical event, the collected information is aggregated as soon as possible. For example, instead of storing the individual time spent on every single call to a specific function, the total time spent on all calls is stored along with the total number of calls to the function. This approach reduces the amount of performance data to be stored dramatically but still allows for drawing the same conclusions as a trace-based analysis: a developer is typically not interested into the time wasted on a single function call but on the total time wasted by a specific function. A function that is called only

[1] **A**utomatic **P**erformance **A**nalysis: **R**eal **T**ools

once during a program run may waste a whole second without any impact on the program's performance. However, a function that is called a million times and wastes just a millisecond on every call usually degrades the performance significantly.

Another benefit of summary information-based performance analysis is the fact that the analysis can be performed online, i.e., while the application is still running. This allows for instantaneous feedback to the user without time-consuming post-processing of trace files after the program has terminated on the one hand. On the other hand, an online analysis allows for reacting on previously gathered information. For example, if a function had been identified as a bottleneck, the analysis tool could drill deeper into the function body to pinpoint actual source code lines that cause the bottleneck.

4.4.2 The Architecture of Periscope

In order to keep pace with the increasing level of parallelism of modern supercomputers and to allow for analyzing large-scale applications with thousands of processes, `Periscope` utilizes a distributed approach for performance analysis. That is, it spawns a hierarchy of communication and analysis agents (see Figure 4.3) for collecting and analyzing the performance-critical data of a parallel application. Each of the analysis agents, i.e., the leaves of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes. The communication agents collect the information gathered by the analysis agents, aggregate it, and propagate it through the hierarchy to the master agent at the top. The master agent connects to the frontend that interacts with the user.

Upon startup, the frontend analyzes the set of processors available, determines a suitable mapping of application processes and agents to the processors, and then starts the application and the hierarchy of communication and analysis agents. Afterwards, a command is propagated from the frontend down to the analysis agents that subsequently start the search for performance bottlenecks. The application's performance data are collected by the application processes themselves by utilizing the `Periscope` performance monitoring library that provides functions for gathering the relevant data. The necessary calls to the library functions are added to the application by instrumenting the application. This is achieved by means of source code instrumentation which allows for selectively instrumenting code regions, i.e., functions, loops, vector statements, OpenMP blocks, IO statements, and call sites. The monitoring library also provides the Monitoring Request Interface (MRI) to which the analysis agents connect. The MRI allows for configuring the measurements and retrieving performance data as well as starting, halting, and resuming the execution of the application.

The foundation of `Periscope`'s performance analysis are so-called *performance properties* that formally represent the performance characteristics of an application. They are specified in the APART specification language ASL [40], translated into C++ classes, and loaded at runtime. An ASL performance property specification consists of three parts: *condition*, *confidence*, and *severity*. *Condition* specifies the condition that has to be met for the property to hold. It can be derived from the performance metrics that are measured by the performance monitoring library.

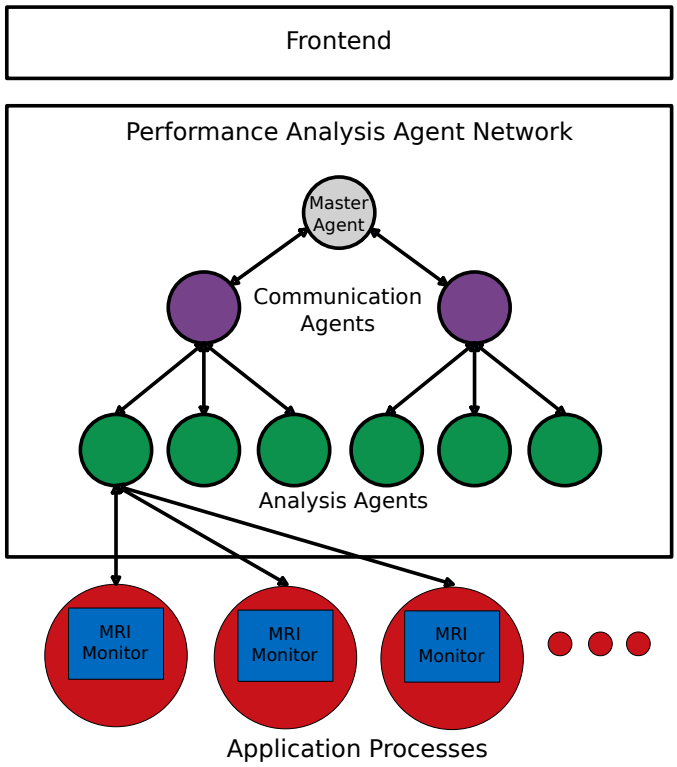

Figure 4.3: `Periscope` consists of a frontend and a hierarchy of communication and analysis agents. The analysis agents configure the MRI-based monitors of the application processes and retrieve performance data.

Confidence is a value in the interval $[0-1]$ that quantifies the certainty that a property holds. *Severity* is an indicator of the importance of the property and of its impact on the application's performance. For example, in case of an MPI application, an interesting performance property is *Early Receiver*. It indicates that a receiving process called the receive function before the corresponding send function has been called and therefore had to wait for the message to arrive. The *condition* of the property can be derived from the waiting time within the MPI receive operation, i.e., if the process had to wait for more than a certain threshold, the property holds. For most properties, the *confidence* is typically 1.0 as they are based on reliable measurements. The *severity* usually relates the time that was wasted to the total runtime of the application. That is, if the waiting time is short compared to the total runtime, the property has only little impact on the application's performance and therefore is not that severe. A collection of relevant OpenMP and MPI performance properties has been assembled by the APART working group [37, 38].

The search for performance bottlenecks is performed in one or more experiments according to a search strategy determined by the user at startup. The search strategies are implemented as C++ classes and custom classes may be provided by the user. Each strategy defines an initial set of hypotheses, i.e., performance properties that are to be checked in an experiment, as well

as the refinement of detected properties to a new set of hypotheses that are to be checked in further experiments. Many applications in HPC have an iterative behavior, e.g., a loop where in each iteration the next time step of the simulated time is performed. If the application has such an iterative behavior, each iteration can be regarded as a single experiment. Otherwise, the whole program is executed for an experiment. The agents start from the initial set of hypotheses, request the necessary information for proving the hypotheses via MRI, release the application for performing the experiment, retrieve the information from the monitor after the processes were suspended again, and evaluate which hypotheses hold. If necessary, a proven hypotheses might be refined and the next experiment is performed. At the end of the local search, the detected performance properties are reported back via the agent hierarchy to the frontend. The communication agents combine similar properties found in their child agents and forward only the combined properties.

The current version of `Periscope` has been adapted to the Intel Itanium2-based HLRB2 supercomputer at the Leibniz Computing Center (LRZ). It allows for both detection of performance bottlenecks limiting the scalability on parallel systems (see next section) as well as analyzing the single-node performance of an application by means of hardware performance counters (see [52] for details). Additionally, it takes the machine topology into account when distributing the analysis agents. It can be used in interactive as well as in batch processing mode.

4.4.3 Identifying MPI Performance Properties

The information required for detecting performance properties related to MPI can be gathered by utilizing the MPI profiling interface (see Section 4.3.2). For profiling purposes one can provide a wrapper for any MPI function that replaces the respective function of the library. It collects the profiling data and provides the actual communication functionality by calling the original function via its 'P'-prepended alias.

Simple metrics that can easily be collected for every function just by counting within the wrapper include the number of calls, the time spent, and the bytes sent or received. In combination with the information from the instrumentation of the source code the acquired profiling data can be associated with each individual call site, i.e., the source file name and the line number of the MPI call. However, the conclusions that can be drawn from such simple metrics are very limited. For example, for many functions the interesting information is not *how much* time has been spent within the function but *for what* it has been spent. Especially for blocking communication operations it is (almost) unimportant how long the actual data transfer took, as this time is typically limited by the network hardware and can hardly be influenced. What matters is the time the process had to wait for a communication partner and thus, the waste of CPU cycles.

Synchronous clocks are a prerequisite to identify and quantify situations where one or several communication partners called their respective MPI functions too late or too early. On systems where synchronous clocks are not provided by the operating system the profiling library needs

to synchronize the clocks of all processes at initialization. This can be achieved by utilizing, e.g., Christian's algorithm [27]. In the following, we assume that all clocks are synchronized.

Whenever a process calls a MPI wrapper function a timestamp is taken at the very beginning of the function. The timestamps of all processes can be compared afterwards to identify the processes that did not call the respective function in time. In general, there are two alternatives to evaluate the timestamps: during runtime immediately after each communication operation or post-mortem by evaluating traces which have been collected during runtime. As mentioned earlier, collecting traces for large-scale applications can generate a huge amount of data that are very challenging to analyze. Since it is `Periscope`'s policy to aggregate information as soon as possible, the timestamps are evaluated immediately after the MPI function call.

To compare the timestamps, at least some of the communication partners have to transfer their timestamps to the other partners. This can easily be achieved by utilizing the respective MPI functions within the wrapper functions. However, one has to make sure to not tamper with the semantics of MPI by doing so. For example, the functions `MPI_Send()` and `MPI_Recv()` are referred to as blocking operations. As indicated in Section 4.3.1, this does not necessarily imply that both functions will only return after the message has successfully been transferred. While this is true for the receive function, the send function usually only blocks until the message has been successfully copied to a send buffer. One can not conclude that the message has actually been received from the fact that the send function has returned. The receive function might not have even been called yet. One has to keep these semantics in mind when orchestrating the exchange of timestamps. Obviously, it is not a good idea to use blocking communication operations to transfer the receiver's timestamp to the sender. This would significantly change the behavior of the send operation: it would block until the actual message has been received and the receiver has sent its timestamp in return. This could influence the program flow considerably and hence render the whole profiling useless. To avoid such interference, timestamps need to be transferred only from the sender(s) to the receiver(s) by utilizing communication operations of the same class as the original operation: `MPI_Send()` for Point-to-Point, `MPI_Bcast()` for 1-to-N, `MPI_Gather()` for N-to-1, and `MPI_Allgather()` for N-to-N. The receiver then calculates the delta between its own timestamp and the sender's timestamp. In case of collective operations, each receiver computes the delta to the timestamp of the first process that called the particular collective function. The delta values are then summed up for each process and each call site. The flowchart in Figure 4.4 shows how the wrapper functions interact by example of `MPI_Send()` and `MPI_Recv()`. By utilizing the MRI interface, the analysis agents can afterwards collect the profiling information from the processes they control and use them for evaluating performance properties.

The fact that profiling information is only gathered by receiving processes and thus only analyzed by their controlling agents leads to the situation that an analysis agent might identify performance properties of a process it does not control. However, this behavior fits well to the distributed approach of `Periscope`. An agent might detect performance properties of a foreign process. Like all other properties it detects, it will propagate them upward through the agent hierarchy where they are aggregated. The master agent at the latest will have all necessary

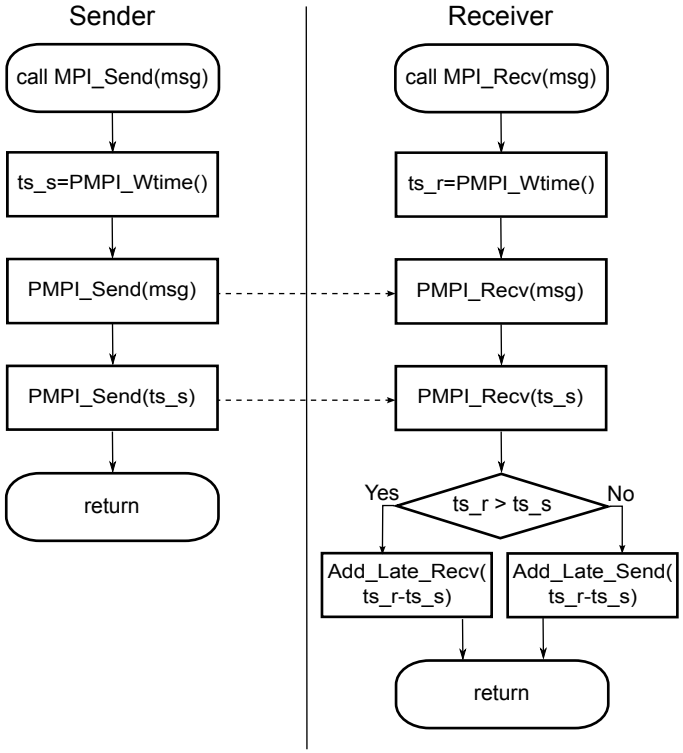

Figure 4.4: Flowchart for `MPI_Send()` and `MPI_Recv()` wrapper functions

information to evaluate the severity of a performance property associated with a specific send operation.

In general, all performance properties that can be found by a post-mortem trace analysis, can also be detected by the strategy described above. More specifically, `Periscope` is able to detect – among others – the same MPI properties that have been defined by the APART working group and implemented for example in KOJAK [155] which is trace-based. By taking the communication semantics of the MPI functions into account, the overhead introduced by the additional communication operations needed for the exchange of timestamps can be kept to a minimum. As a consequence, the influence on the behavior and the program flow of the analyzed application is negligible.

Chapter 5

Parallelization of Phylogenetic Tree Inference Algorithms

This chapter gives an overview of potential sources of parallelism in phylogenetic analyses and how they can be exploited. Additionally, it describes tangible implementations of these parallelization approaches in `RAxML` with OpenMP, PThreads, and MPI.

5.1 Introduction

The computational effort that is required for a phylogenetic analysis under the Maximum Likelihood criterion is mainly influenced by three factors: the size of the input sequence alignment, the number of tree searches to be conducted, and the number of required bootstrap trees.

The size of the input sequence alignment directly influences the computational effort that is required for conducting a single phylogenetic tree search. However, a sequence alignment actually is a two-dimensional data matrix and both dimensions affect the runtime of the tree search differently. As described in Section 2.2, the size of a phylogenetic tree and hence the tree search space grows factorially with the number of species in the tree, i.e., with the number of sequences or lines in the input alignment. Clearly, it is pointless to parallelize an algorithm whose complexity grows factorially. Though, numerous heuristics have been proposed that allow for tree search algorithms with a significantly lower complexity (see Section 2.9 and [94]) and therefore facilitate efficient tools for large-scale tree inference.

The second dimension of the matrix, i.e., the length of the sequences or the number of columns in the input alignment, contributes linearly to the computational effort of a tree search: the length of the (ancestral) conditional probability vectors used for the computation of the likelihood function corresponds directly to the number of distinct sequence patterns in the input alignment. Note that the number of distinct sequence patterns is typically lower than length of the sequences in the input alignment. This is due to the fact that for performance reasons, only a single conditional probability vector entry is used for identical columns of the alignment that is weighted by their number. As the likelihood function essentially is a loop over

the conditional probability vectors, its computational complexity grows linearly with the length of those vectors and consequently with the number of distinct sequence patterns in the input alignment.

The number of tree searches to be conducted depends on the tree search algorithm that is used for the phylogenetic analysis. For an exhaustive tree search algorithm that guarantees to find *the* maximum likelihood tree, a single tree search is sufficient. Yet, due to the previously described computational complexity of such algorithms, they are hardly used for tree searches encompassing more than ten species. As the heuristic algorithms which are used instead cannot guarantee to find the maximum likelihood tree, they require more than a single tree search. The main problem of heuristic search algorithms is that they potentially get stuck in local maxima. Hence, multiple tree inferences are typically started from several distinct starting points (starting trees) in the hope to actually find the global maximum instead of only a local one. The total number of tree searches required to obtain reliable results depends on the search algorithm as well as on the specific input sequence alignment.

Deciding how many bootstrap trees are required for a phylogenetic analysis can be even more difficult (see Section 2.7). The bootstopping criteria proposed by Pattengale et al. [115] can indicate a reasonable value, but in general the rule "the more, the better" applies. Usually it is not so much a question of how many trees are required, but more of how many trees are feasible with regard to available CPU hours.

To sum up, the computational requirements for phylogenetic analyses are mainly driven by the steadily increasing amount of available molecular sequence data. The time required for a single tree inference is a function of the size of the input sequence alignment. However, two different datasets with identical dimensions will not yield identical runtimes as the search path will be different for both datasets. As real-world phylogenetic analyses require multiple tree inferences and bootstraps, the total compute time required for large-scale phylogenetic analysis can easily add up to millions of CPU hours (see [68] for an example). Hence, efficient parallelization approaches need to be devised in order to exploit high performance computing resources that can provide a sufficient amount of computational power. Section 5.2 will provide an overview of potential sources of parallelism in phylogenetic analyses and Section 5.3 will discuss tangible implementations in `RAxML`.

5.2 Sources of Parallelism

Any real-world phylogenetic analysis exhibits several sources of parallelism that can be exploited: *embarrassing parallelism*, *inference parallelism*, and *loop-level parallelism*.

5.2.1 Embarrassing Parallelism

Multiple tree inferences as well as bootstrap analyses required for real-world phylogenetic analyses are embarrassingly parallel. That is, they can be executed independently from each other without interaction. In fact, many users of phylogenetic inference programs intuitively exploit

this inherent parallelism, probably without being aware of this fact: if they have more than one computer available for the analyses, they would start several instances of the program in parallel and later collect the results of the individual program runs. While this may be a viable solution for the rather low number of regular tree searches that are required for a phylogenetic analysis, keeping track of hundreds or even thousands of bootstrap replicates manually is challenging and error-prone and therefore requires automation. In a supercomputing environment, the parallel program runs could be performed by submitting a job for every single tree search to a batch queuing system which would manage the actual program execution. Although this approach relieves the user from the burden of managing computing resources, he still needs to keep track of finished jobs and collect their results. To some extent, these tasks could be automated by utilizing batch or shell scripts. However, as these scripts would need to account for the specific details of the user's computing infrastructure (e.g. operating system, hardware architecture, hostnames, pathnames, etc.) a generic solution would be almost impossible to implement. At least some customization by the user would be required to adapt the scripts to his specific needs. Given the fact that most users of phylogenetic analysis tools have no background in computer science, this task may prevent many of them from exploiting their computational resources efficiently.

Consequently, the exploitation of the inherent parallelism of phylogenetic analyses needs to be implemented at the application level in order to allow for efficient use of parallel computing resources. As the individual tree searches and bootstrap analyses are independent from each other, they can be executed in parallel without interaction and therefore allow for a high degree of parallel efficiency.

5.2.2 Inference Parallelism

Depending on the specific tree search algorithm, an additional source of parallelism may be found at the inference level. For example, many heuristics for phylogenetic tree inference typically involve traversing the tree and modifying its topology by performing rearrangement operations like NNI or SPR. Afterwards, only the branch lengths of the part of tree that was actually affected by the rearrangement operation are re-optimized. Given a large tree topology, rearrangement operations in distant parts of the tree are therefore likely to be independent from each other and could potentially be performed in parallel.

Another example for a potential source of inference parallelism can be found in Bayesian approaches for phylogenetic tree inference which typically employ Metropolis-coupled Markov chain Monte Carlo (MC^3) methods. The individual chains of the Monte Carlo simulation are almost independent from each other and only require interaction when states between two chains are exchanged. Consequently, computing the chains concurrently allows for parallelizing the Monte Carlo simulation. An MPI-based implementation of the widely used Bayesian tree inference tool MrBayes that exploits this source of parallelization has been described by Altekar et al. in [7].

Genetic algorithms exhibit a similar source of parallelism as the individual populations of

those algorithms can be computed independently from each other. Communication between the processes that are computing particular populations is only required for occasionally migrating individuals of one population to another. A parallelization of the genetic algorithm for phylogenetic tree inference as implemented in GARLI has been described by Derrick Zwickl in [163].

However, all sources of parallelism described above highly depend on a specific tree search algorithm and do not allow for a generic parallelization approach. Hence, any modification to the specific search algorithm may render a previous parallelization useless. Additionally, exploiting parallelism at this level may only yield a rather low degree of parallelism. In case of tree rearrangement operations, it can be difficult to determine independent subtrees in advance. Consequently, the parallel efficiency may be limited by hard to resolve data dependencies as modifications in a particular subtree may impact other subtrees that were previously thought to be independent. In case of MC^3-based or genetic algorithms, the maximum degree of parallelism is limited by the number of chains or populations, respectively. Since reasonable values for the number of chains/populations are typically below 8 for most datasets, this approach only allows for a rather low degree of parallelism. It can only be increased, if the number of chains/populations is increased which results in a more thorough coverage of the search space. Although this may lead to a faster convergence of the search algorithm, it is mainly beneficial for the quality for the resulting tree. Additionally, it is difficult to define suitable metrics for assessing the speedup or the benefit of such a parallelization approach as the outcomes of the sequential and parallel computations differ.

5.2.3 Loop-level Parallelism

All phylogenetic analysis tools that employ the phylogenetic likelihood function (PLF) as described in Section 2.6.1 may exploit an additional source of parallelism within the likelihood function itself. Since the evolutionary models used for computing the PLF assume that all nucleotide sites evolve independently of each other, the individual entries of the conditional probability vectors can also be computed independently. Hence, the computation can be performed in parallel and allows for a high degree of parallelism that theoretically corresponds to the length of the probability vectors. As the phylogenetic likelihood function typically accounts for over 95% of total execution time in most programs for phylogenetic tree inference – "classic" Maximum Likelihood-based inference as well as Bayesian inference – it exhibits a generic source of parallelism.

Computing the likelihood score of a fixed tree topology in parallel is straight-forward: given p workers (i.e., threads or processes) and an input sequence alignment consisting of m columns, each worker is assigned a fraction of m/p of the sequence columns and is responsible for computing the corresponding entries of the conditional probability vectors. The only data required for this task are the tree topology, the respective columns of the sequence alignment, the parameters of the model of sequence evolution, and the branch lengths of the tree. After all workers have updated their share of the probability vectors, the log likelihood score of the tree can

be computed by calculating the likelihood scores of the individual sites (Equation (2.19)) and adding up their log values (Equation (2.21)) by means of a parallel reduction operation.

However, in Maximum Likelihood-based inference, the PLF is not only required for computing the score of a fixed tree topology but also for optimizing the branch lengths of the tree as well as the parameters of the evolutionary model (see Section 2.6.3). Most implementations employ iterative optimization methods for this task. That is, in each iteration of the optimization process, a parameter (or branch length) is modified and the likelihood score of the tree is re-computed given the new value of the parameter. Depending on the programming paradigm used for the parallelization of the PLF, the modified parameters may have to be transferred to the workers explicitly before a new iteration can start.

5.3 Parallel Implementations of RAxML

Being a Maximum Likelihood-based tool for phylogenetic tree inference, RAxML exhibits all sources of parallelism described in the previous Section. In the course of time, it has been parallelized in multiple ways, on numerous hardware architectures, and with various software frameworks. Loop-level parallelism has been exploited with OpenMP [141], POSIX threads (PThreads) [139], SIMD instructions [10], and MPI [112] on x86, ia64, and PowerPC architectures, as well as on the Cell Broadband Engine [12] and on graphics cards [24]. Embarrassing parallelism has been exploited with MPI [131] and earlier work focused on parallelization at the inference-level with CORBA [134], a custom http-based protocol [108], and MPI [136]. However, due to the drawbacks described in Section 5.2.2, the work on inference parallelism has been abandoned and we concentrated on generic approaches at the loop-level and on exploting embarassing parallelism instead.

The following sections will only describe those parallel implementations of RAxML in detail, in the development of which I was involved during my PhD studies: OpenMP-, PThreads-, and MPI-based loop-level parallelizations as well as an MPI-based hybrid parallelization that exploits loop-level and embarrassing parallelism simultaneously.

5.3.1 Exploitation of Loop-level Parallelism

The bulk of computations that are related to the phylogenetic likelihood function are concentrated in three functions: `newview()`, `makenewz()`, and `evaluate()`. The `newview()` function computes the conditional probability vector at an internal node from the probability vectors of its two child nodes and the fixed lengths of their connecting branches by means of Equation (2.14). The branch lengths between two nodes are optimized in `makenewz()` by employing the iterative Newton-Raphson method as described in Section 2.6.3.1. The `evaluate()` function computes the log likelihood score of a fixed tree topology by combining the conditional probability vectors of the virtual root's child nodes according to Equations (2.19) and (2.21).

Table 5.1 shows the respective runtime share of `newview()`, `makenewz()`, and `evaluate()` during a single tree search with RAxML on various DNA and amino acid datasets of different

dimensions. The values have been gathered by profiling the sequential version of `RAxML` with OProfile [2] on an Intel Xeon 5150 Woodcrest processor. As can be seen from the table, those three functions account for over 99% of the total runtime regardless of the dimension or the type of data (DNA, protein, etc.) of the input sequence alignment. Concentrating on the parallelization of these functions only should therefore provide good parallel performance of the whole application.

5.3.1.1 OpenMP Parallelization

As the bodies of `newview()`, `makenewz()`, and `evaluate()` essentially consist of `for`-loops over the conditional probability vectors, they are an ideal candidate for an OpenMP-based parallelization. In fact, parallelizing the `newview()` function is as easy as inserting a `#pragma omp parallel for` pre-compiler directive in front of the loop body, followed by a short list of private temporary variables for storing intermediate results. The default scheduling strategy in OpenMP is `static`, i.e., the loop iterations are evenly divided among the threads in contiguous blocks. For example, in case of two threads the first half of the probability vectors will be computed by the first thread, whereas the second half will be computed by the second thread. A graphical representation of this parallelization scheme is depicted in Figure 5.1.

The parallelization of `evaluate()` and `makenewz()` is similar to `newview()` but requires an additional reduction operation at the end of the loops for computing the sum over the log likelihood scores of the individual sites. It is important to emphasize that the use of the reduction operation can be performance critical because it requires synchronization. In case of load imbalance, it will decrease the parallel efficiency of the application as some threads will have to wait at the end of the loop and hence waste CPU cycles. Although the OpenMP runtime will try to avoid load imbalance by distributing the individual loop iterations evenly over all threads, the OpenMP loop parallelization concept is susceptible to it. Due to its very fine-grained nature, even small deviations within the loop body like irregular data access patterns or `if`-clauses can lead to huge runtime differences between particular threads. Changing OpenMP's scheduling strategy from `static` to `dynamic` or `guided` can help to improve load imbalance issues but at the same time induces other problems: both strategies split the iterations into smaller chunks

Dataset	Datatype	# Sequences	# Base-Pairs	`newview()`	`makenewz()`	`evaluate()`	Total
d50_1000	DNA	50	1,000	70.88%	28.06%	0.50%	99.44%
d50_5000	DNA	50	5,000	81.79%	17.57%	0.49%	99.85%
d50_50000	DNA	50	50,000	54.77%	44.01%	1.10%	99.88%
d50_500000	DNA	50	500,000	49.17%	49.83%	0.94%	99.95%
d250_5000	DNA	250	5,000	67.90%	31.39%	0.49%	99.70%
d500_5000	DNA	500	5,000	71.84%	26.91%	1.01%	99.76%
a50_1000	AA	50	1,000	90.31%	9.22%	0.24%	99.77%
a50_5000	AA	50	5,000	79.91%	19.15%	0.83%	99.89%
a250_5000	AA	250	5,000	78.95%	20.01%	0.94%	99.89%
a500_5000	AA	500	5,000	79.36%	19.60%	0.93%	99.89%

Table 5.1: Runtime share of `newview()`, `makenewz()`, and `evaluate()` on a single tree search with various DNA and amino acid sequence (AA) alignments of different dimensions.

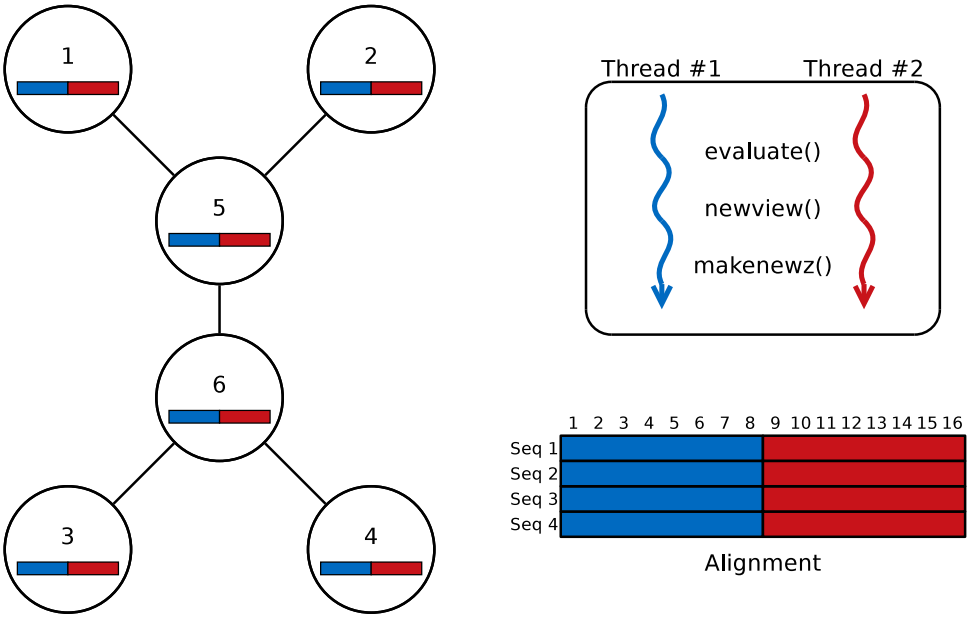

Figure 5.1: Workload distribution scheme of the OpenMP parallelization with two threads. Thread #1 computes the first (blue) half of the probability vectors, Thread #2 computes the second (red) half.

and assign them dynamically to threads which certainly yields a better work load distribution. However, it also induces additional synchronization events that decrease the parallel efficiency. Furthermore, the smaller loop chunks destroy the linear memory access pattern to the conditional probability vectors. Consequently, the CPU's internal memory prefetcher fails to work efficiently, which results in higher memory access latencies and reduced memory bandwidth. Hence, the `static` scheduling strategy still yields the best performance. Yet, it is important to note that in general the reduction operation should be used as rarely as possible and at the most coarse-grained level of parallelism as possible in order to minimize the number of synchronization events.

In addition to the performance problem the reduction operation may cause, its use may also induce numerical problems. Due to cancellation and absorption effects of floating point operations, the order in which arithmetic computations are performed may influence the result of the computations. However, the OpenMP standard defines no order on the arithmetic computations of the reduction operation. Hence, employing OpenMP's internal reduction operation may lead to unpredictable results in case of numerically sensitive algorithms. This is not only a theoretical problem but has in fact caused serious issues in `RAxML` on a large multi-gene alignment. As even small changes to a tree's likelihood score after modifications to its topology have direct influence on the tree search algorithm, any deviation during its computation may result in a totally different search path and hence lead to wrong/non-reproducible results. In order to circumvent such problems, a custom implementation of the reduction operation had to be utilized in order to guarantee a deterministic order of its computations.

5.3.1.2 MPI Parallelization

Although OpenMP is an easy to use approach for parallelizing applications and seems to be especially suited for exploiting `RAxML`'s inherent loop-level parallelism, its major drawback is that it only allows for parallelizing on shared memory systems. Given the cost-effectiveness of small shared memory systems with up to 32 cores, the OpenMP parallelization described in the previous section is especially useful for analyzing datasets that can be handled by those systems in reasonable time. However, for processing datasets that require more cores than that, a distributed memory parallelization is required as machines of that size typically implement a distributed memory architecture (see Chapter 4). Additionally, large datasets require not only plenty of computational power but are also very demanding regarding their memory footprint which could easily exceed 100 GB. Shared memory machines with that amount of memory easily loose their argument of cost-effectiveness and are rarely deployed therefore. Consequently, any parallelization of `RAxML` that shall handle large datasets that require hundreds of processor cores and tens or even hundreds of gigabytes of memory needs to be implemented under the distributed memory programming paradigm. This implies that not only the workload needs to be distributed over a set of parallel processes but also the data-structures that account for the bulk of memory requirements (i.e., the conditional probability vectors).

Parallelization Scheme Exploiting loop-level parallelism in the phylogenetic likelihood function with MPI obviously requires a parallelization scheme similar to the one outlined in Figure 5.1. However, exploiting that kind of data parallelism is much easier under the shared memory programming paradigm than it is under the message-passing paradigm. In a shared memory environment, all data that might be required for any parallel computation can be accessed transparently. Under the message-passing paradigm, however, these data need to be transferred explicitly and – more importantly – timely to the address space of the process that is responsible for performing a particular computation. Yet, the data that are required for computing the PLF are certainly manageable: the molecular sequence data of the OTUs stored at the tips of the tree topology, the branch lengths, and the parameters of the respective model of sequence evolution.

It is important to emphasize, that the tree topology itself is not required for actually computing the phylogenetic likelihood function. Neither `newview()` nor `makenewz()` or `evaluate()` work on a tree structure but only operate on conditional probability vectors and on branch lengths. For example, for computing an internal node's conditional likelihood vector, `newview()` only requires the vector of the internal node, the vectors of its child nodes and the length of their connecting branches. The `evaluate()` function even requires only the probability vectors of the virtual root's child nodes for computing the tree's log likelihood score. The same applies to `makenewz()` that only requires the conditional probability vectors of two nodes for optimizing the length of their connecting branch. However, neither the probability vectors nor the branch lengths need to be stored in a tree structure for those tasks as they are passed to the respective functions only by reference.

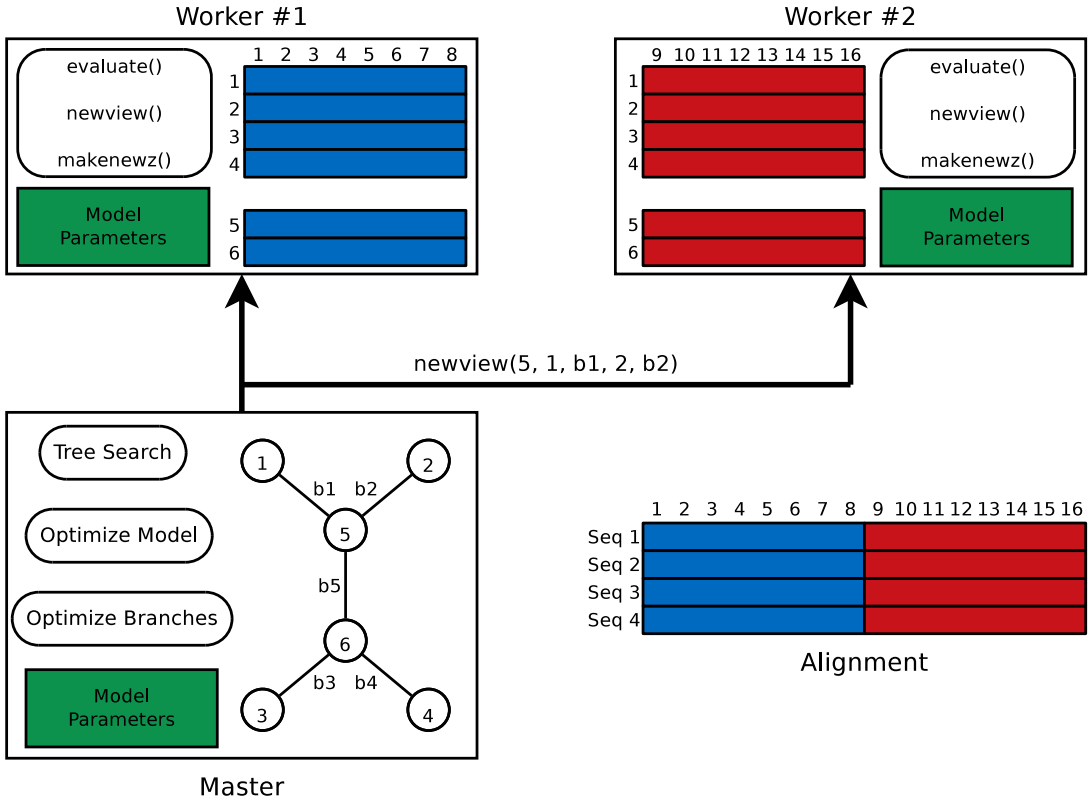

Figure 5.2: Master/worker scheme of MPI parallelization: the conditional probability vectors are distributed evenly over the workers. The master conducts the tree search and steers the branch length- and model parameter-optimization. The actual computations on the conditioal probability vectors are conducted by the workers.

This short list of parameters required for computing the phylogenetic likelihood function and the independence of the individual entries of the conditional probability vector allow for a data-parallel MPI implementation that employs a straight-forward master/worker scheme as outlined in Figure 5.2. The master process holds the tree structure and steers the search algorithm as well as the iterative branch length and model parameter optimization routines. The worker processes are light-weight, i.e., they essentially only implement the functions `newview()`, `makenewz()`, and `evaluate()`. Additionally, they store the conditional probability vectors which are distributed equally over all workers and are enumerated consistently with the master's tree topology. The vectors are computed solely by the workers and are never transferred between the master and the workers. As the data–structures required for the conditional probability vectors typically account for $\sim 90\%$ of the overall memory footprint of `RAxML`, the memory space required for a phylogenetic analysis is equally distributed among the workers. Hence, this parallelization approach allows for overcoming memory requirements as a limiting factor for such analyses.

Due to the fact that the workers implement the actual functionality of the PLF, the master simply has to send short messages to the workers that trigger the respective functions and

contain the required parameters (jobs). Figure 5.2 depicts a sample call to `newview()` which results in the conditional probability vector of node #5 being computed from the vectors of its child nodes #1 and #2 given the lengths $b1$ and $b2$ of their connecting branches.

As all workers execute the same jobs, the `MPI_Bcast()` operation is utilized for sending the messages to the workers. Only `evaluate()` and `makenewz()` return results that need to be transferred back to the master. Since both functions essentially compute sums over the entries of the conditional probability vectors, the partial results of their share of the vectors need to be merged accordingly before they can be delivered to the master. This task can be performed on the fly by the MPI runtime by utilizing the `MPI_Reduce()` operation. Under the assumption that the MPI runtime has been customized to the underlying network hardware and its topology, using solely collective MPI operations like `MPI_Bcast()` and `MPI_Reduce()` guarantees the lowest possible latencies for broadcasting jobs and collecting/reducing results.

Integration Into the Tree Search Algorithm Like most other tree search algorithms, `RAxML`'s algorithm alternates between tree search phases and model optimization phases. During the tree search phase, the model parameters are fixed and the tree's likelihood score is improved by changing the topology and re-estimating only the branch lengths that are immediately affected by the topological rearrangements, i.e., in the neighborhood of topological changes. During the model optimization phase, the parameters of the evolutionary model are re-estimated, and typically the branch lengths of the whole tree are optimized as well.

During the tree search phase, the master and workers continuously execute the following operations: After a change in tree topology the master generates a partial tree traversal list that contains references to the inner nodes' conditional probability vectors that need to be recomputed and then transfers it to the workers which execute the `newview()` operation on the respective vectors. After the vectors have been updated, the master will usually request the workers to execute the `evaluate()` operation in order to compute the tree's likelihood score. The master uses the updated likelihood score to decide whether the change in tree topology will be accepted or whether it should be rejected. A change in the tree topology also requires the execution of `makenewz()` for optimizing the local branch lengths that are most affected by the change.

During the model optimization phase the master will re–estimate the parameters of the model of sequence evolution. Each parameter is optimized via Brent's algorithm [14] which represents an iterative method. After each iteration, the master needs to transmit the current model parameters to the workers, which then recompute *all* conditional probability vectors using the updated model parameters and calculate the new likelihood score of the tree. Once all model parameters have converged towards their maximum, the workers keep local copies of the optimized parameters and use them for further computations of the PLF. As mentioned earlier, the master will not only optimize the model parameters during the optimization phase but will also initiate a global optimization of all branch lengths in the tree. This step is analogous to the branch length optimization during the search phase with the only difference that *all* branches are being optimized.

Note that, the master/worker scheme allows for a higher parallel efficiency than the OpenMP parallelization described in Section 5.3.1.1. This is due to the fact that the computation of conditional probability vectors frequently consists of a series of recursive calls, depending on how many vectors need to be updated due to (local) changes in the tree topology. In order to reduce the communication frequency, such series of recursive calls are transformed into a single iterative sequence of operations by the master which results in a better computation/communication ratio. This approach explicitly makes use of a dependency analysis of the algorithm and reduces the number of synchronization points.

The actual MPI communication between the master and the workers can almost be hidden from the rest of the application as function stubs are provided for `newview()`, `makenewz()`, and `evaluate()` that take the same parameters as the original functions, transfer them to the workers, trigger the actual computation, receive the result, and return it to the caller. Modifications to the source code are essentially only necessary in the program initialization phase for setting up the workers and in the termination phase for shutting down the workers properly. Already this small amount of source code modifications allows for exploiting the loop-level parallelism within the phylogenetic likelihood function and could probably be applied easily to many other tools for phylogenetic tree inference. However, in order to increase the parallel efficiency of this approach, the tree search algorithm had to be modified such that it generates a tree traversal list that comprises all nodes whose conditional probability vectors need to be recomputed instead of calling `newview()` individually for each node.

Accommodating Partitioned Analyses At present, analyses of so-called multi-gene or phylogenomic alignments, i.e., input datasets that comprise concatenated sequence data of several genes, are becoming increasingly popular. Usually such multi-gene analyses are partitioned, i.e., a separate set of likelihood model parameters and branch lengths is estimated for each gene/partition. Biologically this makes sense, since different genes have distinct evolutionary histories.

While the parallelization approach described above has shown to be highly efficient and scalable for unpartitioned analyses, new problems arise for partitioned analyses, as the model parameters and the branch lengths need to be iteratively optimized for every partition separately. The number of iterations required to converge will, in most cases, be different for each partition. An initial, relatively straight-forward approach for dealing with that issue consisted in optimizing the parameters/branches for one partition at a time. For the optimization of the model parameters this is not that critical because a full tree traversal and hence plenty of work is conducted by every worker on its share of the current partition. However, if the branch lengths are optimized on a per-partition basis, the amount of work that can be conducted by every worker in each iteration of the optimization procedure will be rather low. This will result in a bad computation/communication ratio as a synchronizing `MPI_Reduce()` operation is required after each iteration. In the worst case, i.e., a large number of workers and many short partitions with a low number of distinct patterns, it can happen, that there are more workers available than patterns in a specific partition which means that some workers will be idling.

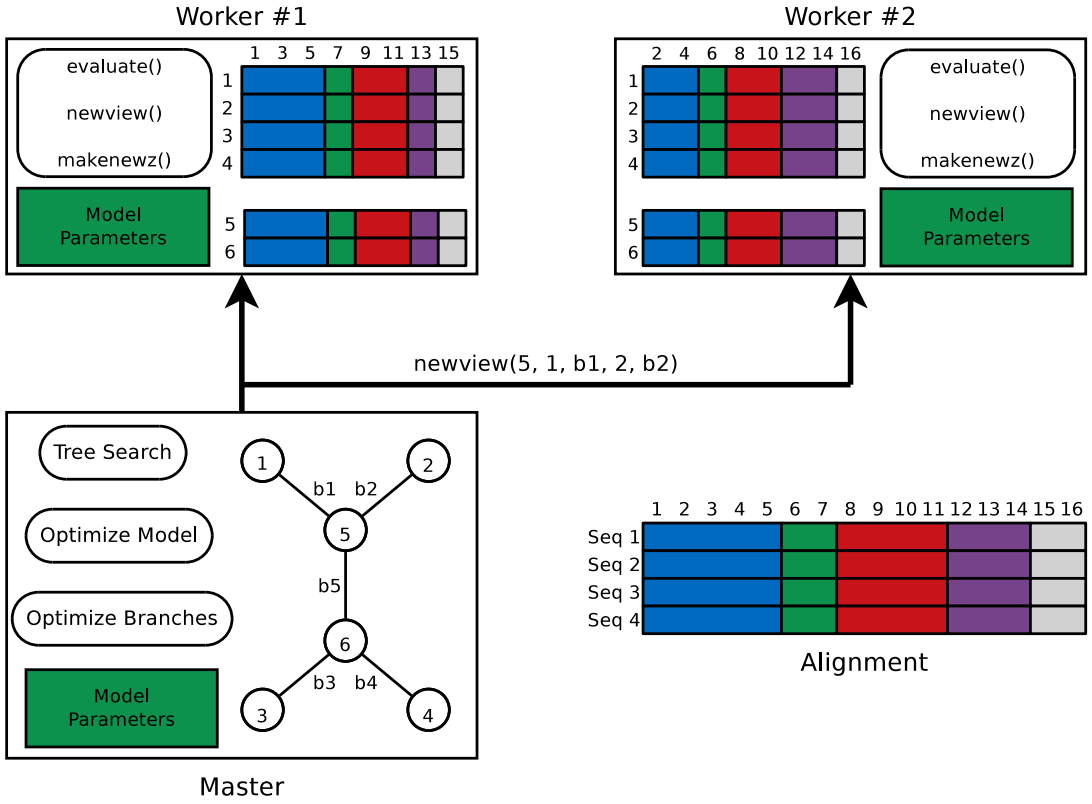

Figure 5.3: Parallelization for partitioned analyses: the conditional probability vectors of each partition are distributed round-robin over the workers.

Consequently, handling partitioned analyses efficiently requires two major modifications to the previously described parallelization approach. Most importantly, the distribution scheme of the conditional probability vectors needs to be changed. Instead of distributing the vectors in consecutive chunks to the workers, they need to be distributed in a cyclic round-robin fashion as depicted in Figure 5.3. Given p workers, worker #1 will hold the columns $\{1, p+1, 2p+1, ...\}$, worker #2 $\{2, p+2, 2p+2, ...\}$, and so on. As the individual entries of the conditional probability vectors can be computed independently of each other and since the reduction operations that are performed on these vectors are commutative, this modification will not alter the results of the computations (at least in theory; in practice, the result may differ slightly due to cancellation and absorption effects of floating point operations). Yet, it guarantees that every worker will have an approximately balanced portion of columns of each partition and hence facilitates load balance. This applies especially to partitioned analyses of mixed DNA and protein datasets: the computation of a conditional probability vector entry for a protein pattern position requires significantly more floating point operations, since there are 20 instead of 4 likelihood entries to compute.

The second modification that was required for handling partitioned analyses resulted in a complete re-design of the iterative optimization procedures in `RAxML` in order to conduct

computations on the full length of the conditional probability vectors over all partitions for as long as feasible, to provide as much work as available to the worker processes, and reduce the synchronization overhead. The basic idea consists in modifying the routines that implement the Newton-Raphson and Brent procedures in such a way that they simultaneously optimize all partitions. Because the iterative optimization procedures on every partition will converge after a variable number of iterations, the master needs to keep track of the convergence conditions for every partition separately and hence avoid the invocation of the likelihood functions on partitions that have already converged via an appropriate boolean vector. While this approach sounds relatively straight-forward it was a major software engineering challenge due to the code complexity of `RAxML` that can handle concatenated datasets consisting of morphological, DNA, multi-state, secondary structure, and protein data.

5.3.2 PThreads Parallelization

From a programmer's point of view, using OpenMP for parallelizing an application is easier and requires less programming effort than using POSIX threads directly by means of the PThreads library. However, from the user's point of view, OpenMP has a significant drawback: it requires an OpenMP-enabled compiler. Especially for an application like `RAxML` that is distributed in source code and the user base of which mostly consists of non-computer experts, this requirement effectively keeps most users from employing the parallel version. In contrast to that, a multi-threaded application that is solely based on PThreads can be compiled out-of-the-box on almost all Linux/Unix systems with the GNU `gcc` compiler that ships with most Unix-style operating systems. Although newer versions of `gcc` are capable of compiling OpenMP applications, the majority of installed Unix systems probably still uses an older version that lacks OpenMP support. Consequently, in order to enable a bigger part of `RAxML`'s users to exploit their shared-memory systems, a PThreads-based parallelization has been implemented.

An intuitive approach to exploit the loop-level parallelism within the phylogenetic likelihood function with PThreads would probably be to copy the parallelization scheme of OpenMP. That is, to extract the loops of the PLK into separate functions, fork threads for executing several instances of those loop-functions in parallel, and assign each instance an equally large share of the loop iterations. However, this approach combines solely the drawbacks of both OpenMP as well as PThreads programming: the limitation to parallelizing loop bodies only and the complexity of the PThreads programming interface.

Probably a better approach for a PThreads-based parallelization would be to adapt the master/worker scheme of the MPI parallelization accordingly. One of the main benefits of this scheme certainly is its increased parallel efficiency which is due to the fact that the computations of multiple conditional probability vectors can be aggregated into a single job. As described in the previous section, this allows for reducing the number of synchronization events. Although synchronization is more costly in message-passing environments, it still can be an issue in shared memory systems and reducing the number of such events evidently is beneficial for any parallelization paradigm.

Besides the algorithmic advantages of the master/worker parallelization scheme, an additional benefit is the fact, that a PThreads-based implementation can easily be co-developed with an MPI implementation. This property is mainly due to the conceptual similarity of processes, which are the building primitives of MPI, and threads: the encapsulation of the master and the worker functionality into individual processes can be applied analogously with threads and requires only little modification. Yet, the main difference between threads and processes is the way they interact. While processes need to explicitly exchange messages due to the lack of a shared address space, threads can communicate with each other implicitly by writing and reading to/from shared variables. However, the message-based communication can easily be "simulated" in a shared memory environment by designating shared buffers into which senders write their messages and from which receivers read.

Consequently, the MPI parallelization of `RAxML` could be transformed to PThreads with reasonable effort. Nevertheless, in order to increase the parallel efficiency on shared memory architectures the PThreads version differs slightly from the MPI version as the PThreads master also implements the worker functionality. That is, it conducts an equally large part of computations on the conditional probability vectors like any other worker. As the PThreads implementation is mainly designed for multi-core computers and smaller shared-memory systems on which the master needs to handle only a few workers it would be idling most of the time otherwise. The MPI implementation, however, is designed for large supercomputers and for handling hundreds of workers. Hence, the MPI master performs no computations but only orchestrates the workers in order to not become a bottleneck.

Deriving one parallel implementation from the other also has the benefit that the PThreads version automatically inherits the MPI version's capability of handling partitioned analyses efficiently which the OpenMP implementation was lacking.

5.3.3 Simultaneous Exploitation of Loop-level and Embarrassing Parallelism with MPI

As indicated in Section 5.2.1, exploiting embarrassing parallelism is rather straight-forward as multiple tree searches or bootstraps can be conducted independently in parallel. The initial MPI-based parallelization of `RAxML` [131] utilized a master/worker scheme for exploiting embarrassing parallelism, in which the master process also acted as a worker but was additionally responsible for assigning tree search jobs to the individual workers and for collecting their results.

However, as the input sequence alignments continue to grow, an additional level of parallelism is required within the workers in order to finish a single tree search within reasonable time and hence allow for short turnaround times. This additional level of parallelism can be exploited at the loop-level as well as the inference level. Yet, due to the previously described advantages of exploiting the loop-level parallelism within the phylogenetic likelihood function, only a scheme for simultaneous exploitation of loop-level and embarrassing parallelism will be described here. Given a stable OpenMP- or PThreads-based parallelization, an additional MPI

layer can be added in order to facilitate a multi-level parallelization. Extending an OpenMP-based parallelization will typically be easier as it concentrates only on compute-intensive loops and is mainly based on pre-compiler directives that will not interfere with the MPI routines. Nevertheless, a hybrid PThreads/MPI parallelization is feasible as well and has in fact been implemented by Pfeiffer and Stamatakis for RAxML [118].

Evidently, the same limitations that have previously been described for the pure OpenMP- and PThreads-based parallelizations of RAxML, also apply to a hybrid implementation that utilizes shared memory programming for exploiting loop-level parallelism: firstly, the maximum degree of parallelism for a single tree search or bootstrap is limited by the number of cores within a shared memory node. Secondly, the maximum size of the input sequence alignment is limited by the available main memory per node. Consequently, the capacity of a single node determines the maximum size of the input dataset although the total number of available processor cores as well as their aggregated amount of memory would allow for significantly larger datasets.

In order to overcome these limitations, MPI has to be utilized for exploiting both levels of parallelism, loop-level as well as embarrassing parallelism. However, adding an additional layer for exploiting embarrassing parallelism to the existing MPI-based loop-level parallelization as described in Section 5.3.1.2 is not as straight-forward as it is for the OpenMP or PThreads parallelization. As the MPI-based loop-level parallelization already employs a master/worker scheme, applying an additional master/worker scheme for exploiting embarrassing parallelism is challenging. In order to facilitate such a parallelization approach, one master process has to assume the role of a super-master that orchestrates the individual tree searches or bootstraps of all other masters. In other words, the remaining masters can be considered as the workers of the super-master. Each master – including the super-master – conducts independent tree searches and steers the likelihood computations of its private set of workers. The results of all tree searches are collected by the super-master. A sample scenario with a total number of 16 processes divided into four subgroups – consisting of a sub-master and three workers – is depicted in Figure 5.4.

The implementation in RAxML works as follows: Initially, the MPI_COMM_WORLD communicator is divided into custom communicators such that each subgroup has its own communicator that only comprises the processes that belong to the respective subgroup. This step is required for handling the collective operations like MPI_Bcast() and MPI_Reduce() correctly within each subgroup. Usually the MPI_Comm_split() command is used for creating custom communicators. However, on the IBM BlueGene/L supercomputer, special care needs to be taken on shaping the communicators. This is due to the fact, that collective MPI operations can utilize the BlueGene's specialized low-latency network for collective communication only if the MPI_COMM_WORLD communicator is used. When custom communicators are used, collective communication is handled over the point-to-point network which has a higher latency. Hence, the MPI_Cart_create() and MPI_Cart_sub() operations are utilized instead of MPI_Comm_split() in order to build each sub-communicator as a 3D mesh with dimensions x, y, z, such that $x \approx y \approx z$. The master node has coordinates (0,0,0). This mechanism is well–adapted to the collective operations implementation in IBM MPI, as the exact algorithm for those operations depends on the message size

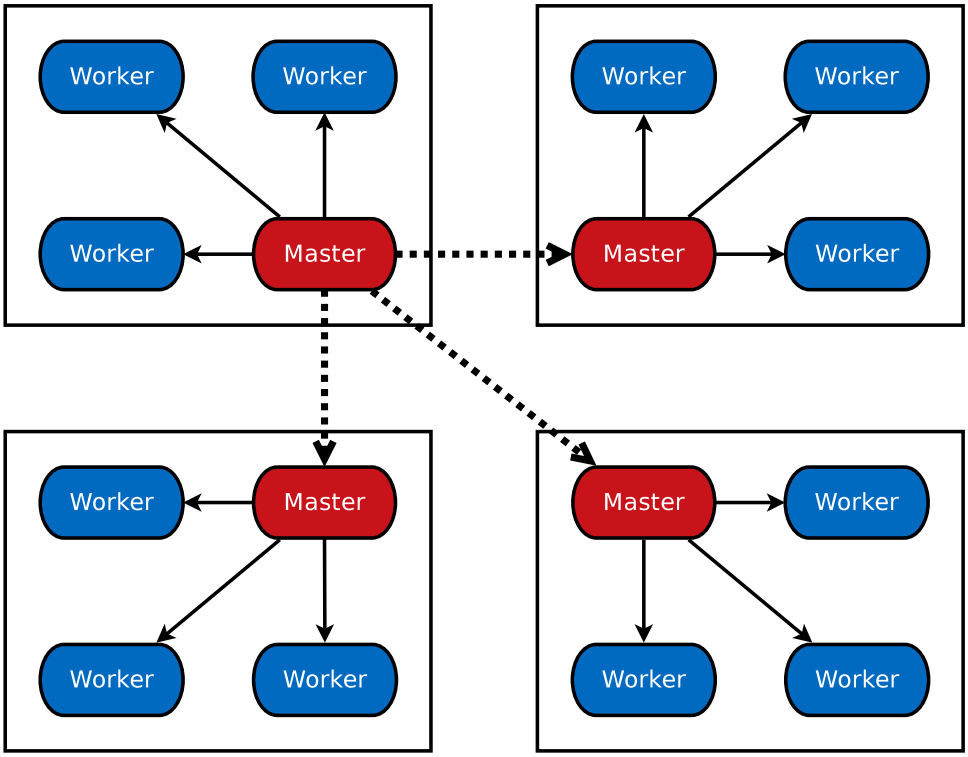

Figure 5.4: Hybrid MPI/MPI parallelization scheme for simultaneous exploitation of loop-level and embarassing parallelism. The super-master orchestrates the individual tree searches of the sub-masters. Each sub-master steers the likelihood computations of its workers.

and the shape of the communicator, e.g., if it is rectangular. This shape of the communicator guarantees that optimal algorithms, which, e.g., benefit from deposit bits, are employed on the point-to-point network. Consequently, the latency is only slightly higher compared to the latency of the `MPI_COMM_WORLD` communicator using the specialized collective network (3.35 μs, plus 90 ns per hop versus 2.5 μs).

Once the communicators have been set up the master with MPI rank 0 becomes the super-master. At program initialization, each master process immediately starts computations on bootstrap replicates or ML searches without communicating with the super-master. Every time a master has completed the computations on a tree it stores the tree locally in a list and sends a message to the super-master. This message contains the number of trees that have been computed so far by this specific master. Every time the super-master receives such a message it checks if the total number of trees specified by the user has already been computed. If that is the case, the super-master sends a termination message to all other master processes. When a master receives the termination message it sends all locally stored trees to the super-master, which prints them to file. Thereafter, each master terminates along with the respective worker processes. When all tree topologies have been written to file, the super-master exits as well. This algorithm avoids the perturbation of fine-grained work scheduling at the super-master, since the

actual tree topologies are only sent at the end of the computation. In theory it could occur that more trees than requested by the user are computed if two masters finish tree computations at the same time. However, such a case has not been observed in our experiments because the probability of such an event is low as the individual masters take different paths through the huge search space. In addition, the computation of potentially more trees than requested by the user within the same amount of time does not represent a disadvantage. The check for pending master/super-master communication messages has been integrated into the `newview()` function as it is by far the most frequently invoked ML function and thereby provides a fine enough granularity to rapidly detect master/super-master communication requests.

Chapter 6

Evaluation

This chapter describes the setup and the results of the experiments that have been conducted in order to assess the performance of the parallel implementations of `RAxML` described in Chapter 5.

6.1 Experimental Setup

For assessing the performance of the parallel implementations of `RAxML`, two large and challenging real-world datasets have been chosen:

1. A multi-gene alignment of 2,182 mammalian sequences with 51,089 base-pairs that comprises data from 67 different genes. Large-scale analyses of mammalian phylogenies have recently received considerable attention (including the popular press), since they can be used, e.g., to date the rise of present-day mammals [11].

2. A matrix of 566,470 polymorphic data points for 270 taxa. This dataset contains genotype data for 283,235 non-redundant SNPs on the human chromosome 1 in sorted order for 270 unrelated individuals in the HapMap project [146]. Each column in the matrix is occupied by nucleotides A,C,G,T. For each SNP each individual contributes a pair of consecutive nucleotide columns (1 each from their paternal and maternal chromosome, sorted alphabetically within each pair). Thus, for 283,235 SNPs the total sequence alignment length is 566,470 base-pairs.

In order to test the scalability on various dataset-sizes we also used appropriate sub-alignments that have been extracted from the above datasets. The dimensions and memory footprints of

Dataset	Datatype	# Sequences	# Base-Pairs	# Patterns	Footprint
d50_50000	DNA	50	50,000	23,385	137 MB
d50_500000	DNA	50	500,000	216,025	1,266 MB
d250_500000	DNA	250	500,000	403,581	12,217 MB
d500_5000	DNA	500	5,000	3,829	232 MB

Table 6.1: Properties of alignment sub-samples. The memory footprints refer to the GTR+Γ model.

these sub-alignments are listed in Table 6.1. Note that the table lists the dimensions of the alignments as well as the corresponding number of distinct sequence patterns (see Section 5.1 for an explanation). The datasets d50_50000 and d500_5000 have been extracted from the mammalian dataset, d50_500000 and d250_500000 from the HapMap dataset.

A variety of shared memory and distributed memory architectures of different size, ranging from workstations to clusters and supercomputers, has been used for assessing the performance of the parallelization approaches. On the workstation-scale, the following multi-core systems have been used:

- **K8:** An 8-way AMD Opteron system equipped with Santa Rosa dual-core processors, 16 cores total. Processor model 8218, K8 microarchitecture, 2.6 GHz, 5.2 GFLOPS/core, 2x 1 MB L2 Cache. 64 GB DDR2-667 memory, 9.9 GB/s per socket.

- **K10:** A 2-way AMD Opteron system equipped with Barcelona quad-core processors, 8 cores total. Processor model 2352, K10 microarchitecture, 2.1 GHz, 8.4 GFLOPS/core, 4x 512 kB L2 Cache, 2 MB L3 cache. 16 GB DDR2-667 memory, 9.9 GB/s per socket.

- **Core:** A 2-way Intel Xeon system equipped with Clovertown quad-core processors, 8 cores total. Processor model X5355, Core microarchitecture, 2.66 GHz, 10.6 GFLOPS/core, 2x 4 MB L2 Cache. 8 GB DDR2-1066 memory, 7.9 GB/s total.

- **Nehalem:** A 2-way Intel Xeon system equipped with Gainestown quad-core processors, 8 cores total. Processor model X5550, Nehalem microarchitecture, 2.66 GHz, 10.6 GFLOPS/core, 4x 256 kB L2 Cache, 8 MB L3 Cache. 12 GB DDR3-1333 memory, 29.8 GB/s per socket.

On the HPC-scale, the following clusters and supercomputers were available for the experiments:

- **TUM-IC:** The InfiniBand Cluster at Technische Universität München. A custom Linux cluster consisting of 32 AMD Opteron 4-way nodes equipped with SledgeHammer single-core processors, 128 cores total. Processor model 850, K8 microarchitecture, 2.4 GHz, 4.8 GFLOPS/core, 1 MB L2 Cache. 8 GB DDR-400 memory, 6.1 GB/s per socket. 4x SDR Infiniband interconnect.

- **BGL:** The BlueGene/L system at Iowa State University. A one–rack machine consisting of 1,024 System-on-Chip 1-way nodes with IBM PowerPC dual-core processors, 2,048 cores total. Processor model PPC440, 700 MHz, 2.8 GFLOPS/core, 2x2 kB L2 Cache, 4 MB L3 Cache. 512 MB DDR-350 memory, 5.2 GB/s per node. Multiple proprietary interconnects.

- **HLRB2:** The national supercomputer HLRB2 at Leibniz Rechenzentrum. An SGI Altix 4700 system consisting of 4,096 Intel Itanium2 nodes (3,328 1-way and 768 2-way) equipped with Montecito dual-core processors, 9,728 cores total. Processor model 9040, Itanium microarchitecture, 1.6 GHz, 6.4 GFLOPS/core, 2x256 kB L2 Cache, 2x9 MB L3 Cache. 8-16 GB DDR2-533 memory, 7.9 GB/s per node. NUMAlink 4 interconnect.

- **ICE:** The SGI ICE cluster at Leibniz Rechenzentrum. An SGI Altix ICE system consisting of 48 Intel Xeon 2-way nodes, equipped with Intel Gainestown quad-core processors, 384 cores total. Processor model E5540, Nehalem microarchitecture, 2.53 GHz, 10.1 GFLOPS/core, 4x256 kB L2 Cache, 8 MB L3 Cache. 24 GB DDR3-1066 memory, 23.8 GB/s per socket. 4x DDR Infiniband interconnect.

Note that the identifiers in bold letters at the beginning of each system's description will be used for the remainder of this chapter for referring to the particular systems.

On BGL, the IBM XL C/C++ Advanced Edition for BlueGene compiler suite version 8.0 has been used for compiling the MPI version of `RAxML`. On all other architectures, the Intel compiler suite version 10.1 has been used for compiling all program versions. On each platform, the platform-specific compiler optimizations flags used were identical for all versions with only one exception: on the HLRB2, interprocedural optimizations (IPO) caused a performance degradation by a factor of 3 if applied to the sequential version. These optimizations have therefore been disabled in that case. For all other compilations IPO was enabled as it slightly improved performance.

All runtime measurements have been conducted under the GTR+Γ model. A fixed Maximum Parsimony starting tree has been used for all program runs in order to obtain reproducible results as this disables the randomness in `RAxML`'s tree search algorithm. Each experiment has been conducted at least three times and all execution times reported here are the respective median values of those experiments. For the multi-core platforms only the best runtimes are reported, i.e., for every number of cores used, an optimal thread pinning has been chosen for the experiment.

6.2 OpenMP

The performance of the OpenMP parallelization of `RAxML` has been assessed on the multi-core workstations *K8, K10, Core,* and *Nehalem* as well as on the *HLRB2* supercomputer with the *d50_50000* and *d500_5000* datasets. Both datasets differ significantly in their computation to communication ratio, i.e., this ratio is approximately 100 times less favorable for *d500_5000*. Compared to *d50_50000*, the memory footprint of dataset *d500_5000* is about twice as high. Absolute execution times and speedup values for both datasets on all five platforms are given in Table 6.2. Please note that the sequential version of `RAxML` has been used for measuring the runtimes with a single thread. Figure 6.1 shows plots of the runtimes for the *d50_50000* dataset, Figure 6.2 the corresponding speedups. The respective plots for the *d500_5000* dataset are depicted in Figures 6.3 and 6.4.

The sequential runtimes for the *d50_50000* dataset essentially reflect the expectations one might have from the individual architectures given their properties according to the datasheets: the Nehalem system outperforms all other systems due to its high peak performance, large last level cache, and unrivaled memory bandwidth. The inferior memory subsystem of the Core microarchitecture – with regard to the cache sizes as well as the memory bandwidth – results in

a 33% higher runtime compared to the Nehalem system although both microarchitectures are very similar otherwise. The similarity in the memory subsystems of the AMD processors yields almost identical execution times for the K8 and K10 system which are, however, approximately 57% higher than on the Core system and can be attributed to the smaller caches and the lower peak performance of the AMD processors. The poor performance of the HLRB2 system can mainly be explained by the comparatively low peak performance of the Itanium processor and the low memory bandwidth. Additionally, as the Itanium processor implements a VLIW architecture, its application performance depends highly on the quality of the utilized compiler. In fact, previous experiments with version 9.1 of the Intel compilers yielded execution times that were up to 100% longer as the times reported here for version 10.1 on the HLRB2. Evidently, the quality of the Intel Itanium compilers improved significantly from one generation to the next. However, similar effects have not been observed for the x86 architectures.

All architectures yield super-linear speedups with 2 and 4 threads due to increased cache efficiency: given the rather small memory footprint of the *d50_50000* dataset, the aggregate cache size of an increasing number of cores allows for storing larger portions of the working set in the caches and hence reduces the number of expensive main memory accesses. Note that similar effects have also been observed in previous experiments [141]. The 8-way K8 system even shows super-linear scaling with 8 and 16 threads. Thanks to its balanced NUMA architecture, the aggregate memory bandwidth grows with the number of utilized cores and therefore is no bottleneck for its scalability. The 2-way NUMA architectures of the K10 and the Nehalem system deliver the maximum memory bandwidth with two threads already and hence their speedup levels with 8 threads that saturate the total memory bandwidth completely. For the FSB-based Core system the per-thread memory bandwidth decreases with every additional thread. Evidently, the memory bandwidth is nearly saturated by four threads and therefore almost no speedup can be gained from additional threads. The HLRB2 shows poor scaling behavior for more than 4 threads although it is a NUMA architecture and despite the large last level caches of its Itanium processors. This is due to the fact that the share of conditional probability vectors per thread decreases with an increasing number of threads. Consequently, the compu-

Dataset	#	K8		K10		Core		Nehalem		HLRB2	
		Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp
d50_50000	1	5295		5335		3382		2533		5801	
	2	2093	2.53	2113	2.53	1454	2.33	955	2.65	2446	2.37
	4	1101	4.81	1152	4.63	821	4.12	487	5.20	1263	4.59
	8	617	8.59	714	7.47	707	4.79	289	8.75	917	6.32
	16	315	16.79							488	11.87
d500_5000	1	26467		23278		14815		13623		28828	
	2	8562	3.09	8901	2.62	6024	2.46	5528	2.46	13215	2.18
	4	4242	6.24	4498	5.17	4691	3.16	2968	4.59	10370	2.78
	8	2699	9.81	2716	8.57	2853	5.19	1671	8.15	8437	3.42
	16	2618	10.11							8331	3.46

Table 6.2: Runtimes and speedups (columns Time and SpUp) of the OpenMP parallelization for 1-16 threads (column #) on the K8, K10, Core, Nehalem and HLRB2 systems.

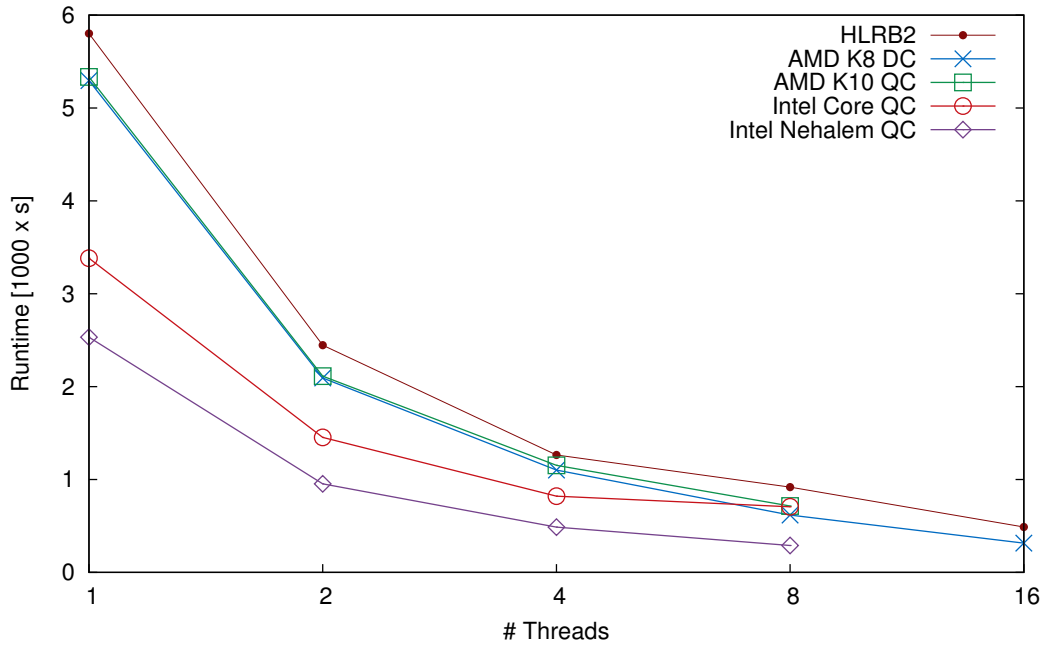

Figure 6.1: Execution times of the OpenMP parallelization on the d50_50000 dataset.

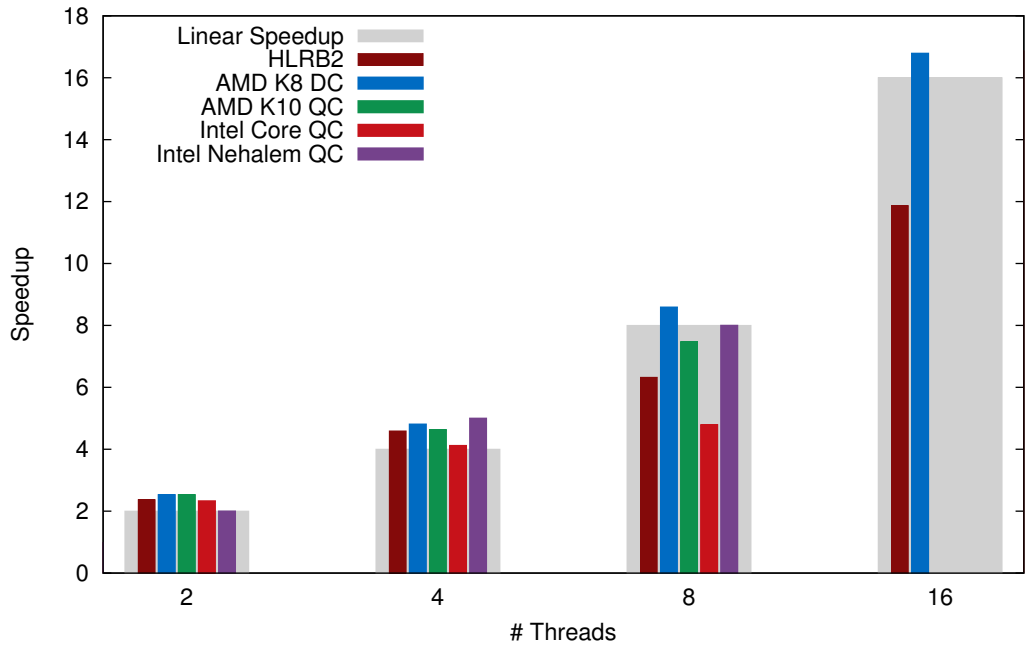

Figure 6.2: Speedups of the OpenMP parallelization on the OpenMP d50_50000 dataset.

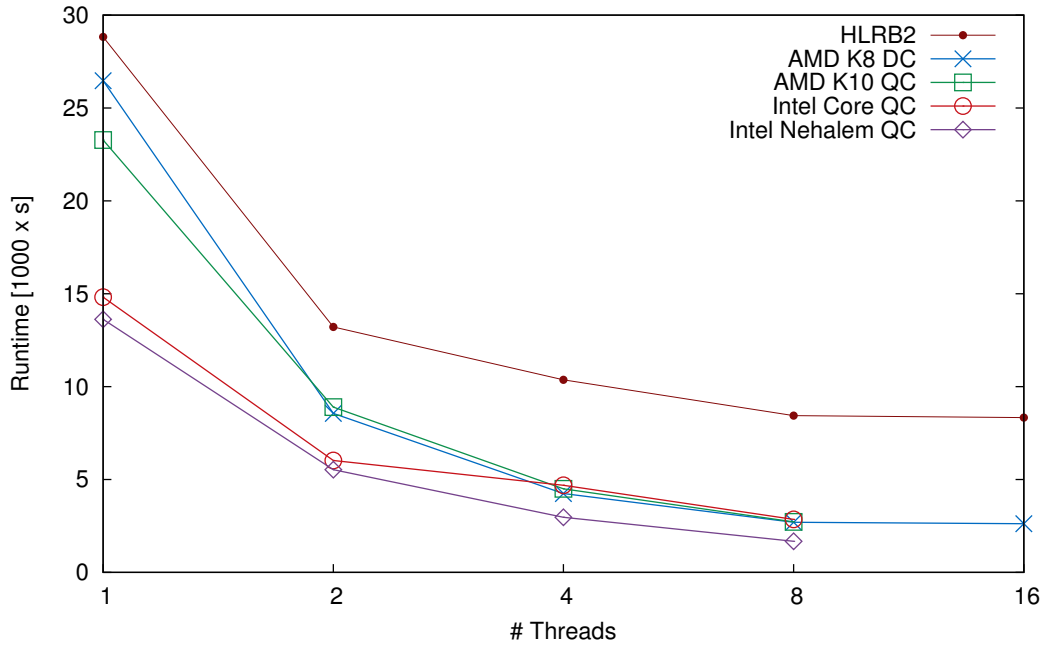

Figure 6.3: Execution times of the OpenMP parallelization on the OpenMP d500_5000 dataset.

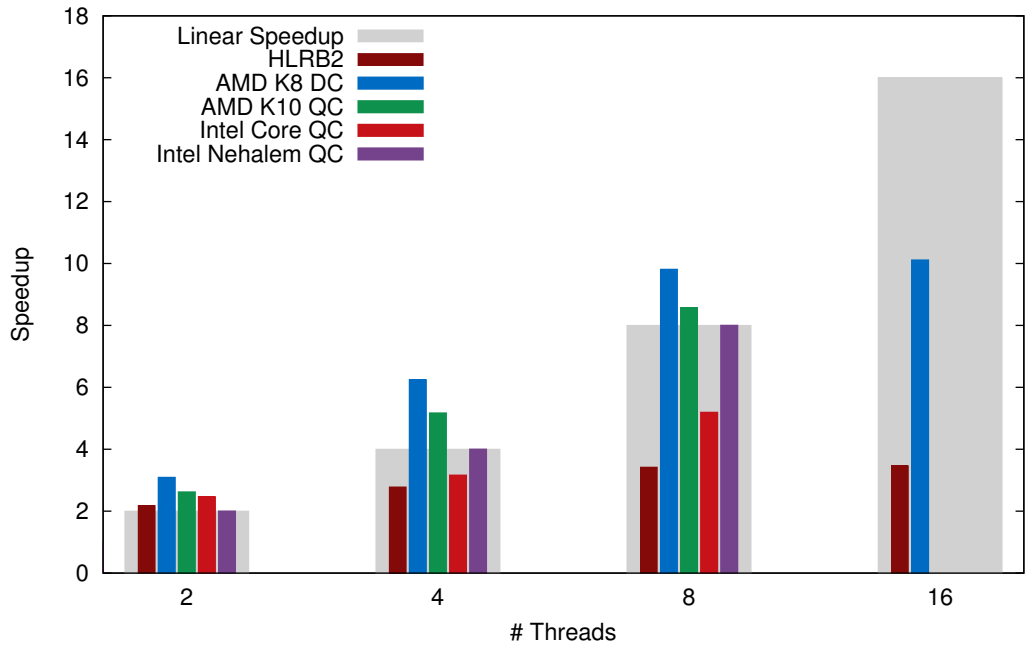

Figure 6.4: Speedups of the OpenMP parallelization on the OpenMP d500_5000 dataset.

tation/communication ratio decreases which results in a higher frequency of synchronization events as all threads need to synchronize after they have computed their share of probability vectors. Since the HLRB2 implements a DSM architecture, synchronization operations need to be performed over the network which impacts performance and becomes increasingly expensive with an increasing number of threads.

For the *d500_5000* dataset the performance ranking of the hardware platforms is identical to the *d50_50000* dataset. There are, however, slight shifts with regard to the performance advantages of particular systems: while the Nehalem system clearly outperforms the Core system in case of the *d50_50000* dataset, their runtimes are almost identical for *d500_5000*. The significantly higher memory bandwidth of the Nehalem system does not seem to have similar impact on the performance as it had in case of the *d50_50000* dataset. Given the similarity of both systems' microarchitectures, the runtime difference can therefore be attributed to the bigger last level cache of the Nehalem processor that facilitates the larger tree data structures of the *d500_5000* dataset. The same applies to the increased runtime difference between the K8 and K10 system.

Due to the 100 times less favorable computation/communication of the *d500_5000* dataset, the HLRB2 system shows poor scaling behavior for more than 2 threads. Again, this can be explained by the expensive synchronization operations that are usually performed over network. However, as the program runs with 2 threads have been carried out on a single node, the synchronization operations between both threads could be performed over shared memory at much lower latency in this case. Consequently, a slightly super-linear speedup could be achieved. All x86 NUMA architectures (K8, K10, and Nehalem) show super-linear speedups for up to 8 threads. Since these systems are true shared memory architectures, the performance impact of the increased synchronization frequency seems to be over-compensated by the increased cache efficiency. However, neither for the K8 system nor for the HLRB2 any significant runtime improvement can be achieved by doubling the number of threads from 8 to 16. Obviously the higher costs of each particular synchronization operation as well as their increased frequency prevent better scalability in these cases. Again, the performance of the Core system suffers from the limited memory bandwidth of its FSB-based architecture which results in clearly sub-linear speedups for more than 2 threads.

6.3 PThreads

The same datasets and systems that have been used for assessing the performance of the OpenMP implementation have been used for evaluating the PThreads implementation as well. The absolute runtimes and speedups for both datasets are given in Table 6.3. For the single thread runs the runtimes of the sequential version of `RAxML` are given again. Hence, those times are identical to the times given in table 6.2. The runtime and speedup plots for the *d50_50000* dataset are depicted in Figures 6.5 and 6.6. The respective plots for the *d500_5000* dataset are shown in Figures 6.7 and 6.8.

The global picture of the runtimes and consequently of the speedups looks similar to the

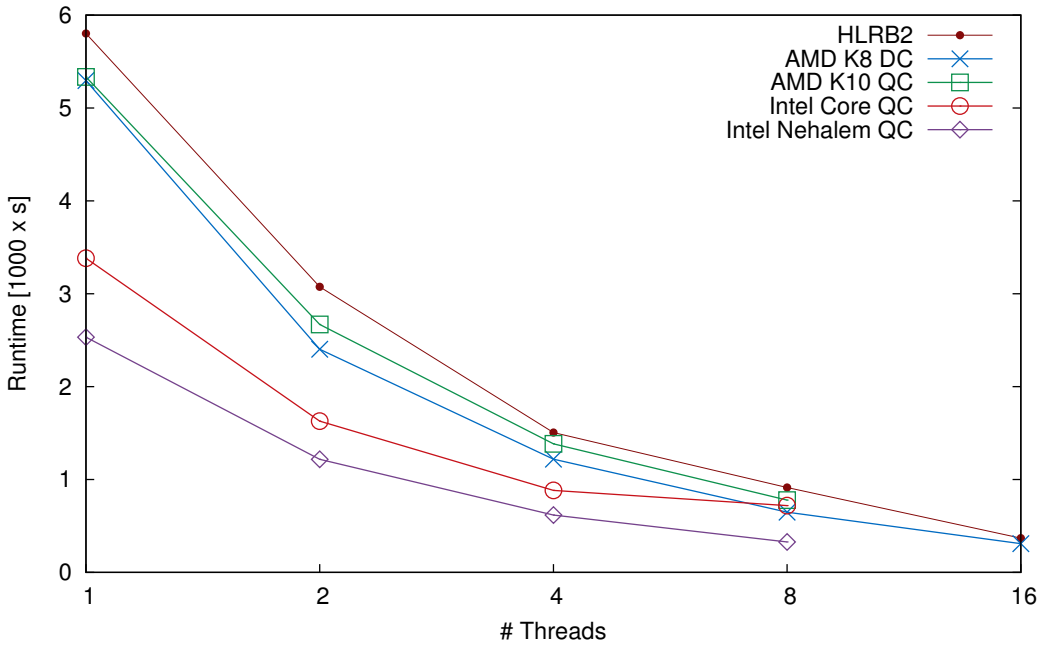

Figure 6.5: Execution times of the PThreads parallelization on the d50_50000 dataset.

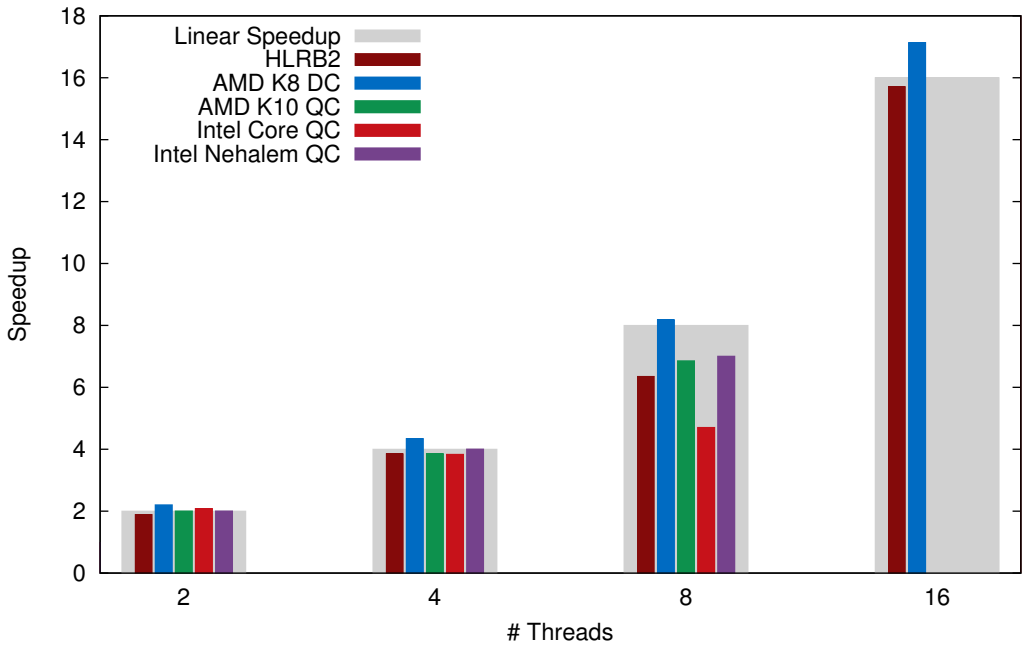

Figure 6.6: Speedups of the PThreads parallelization on the d50_50000 dataset.

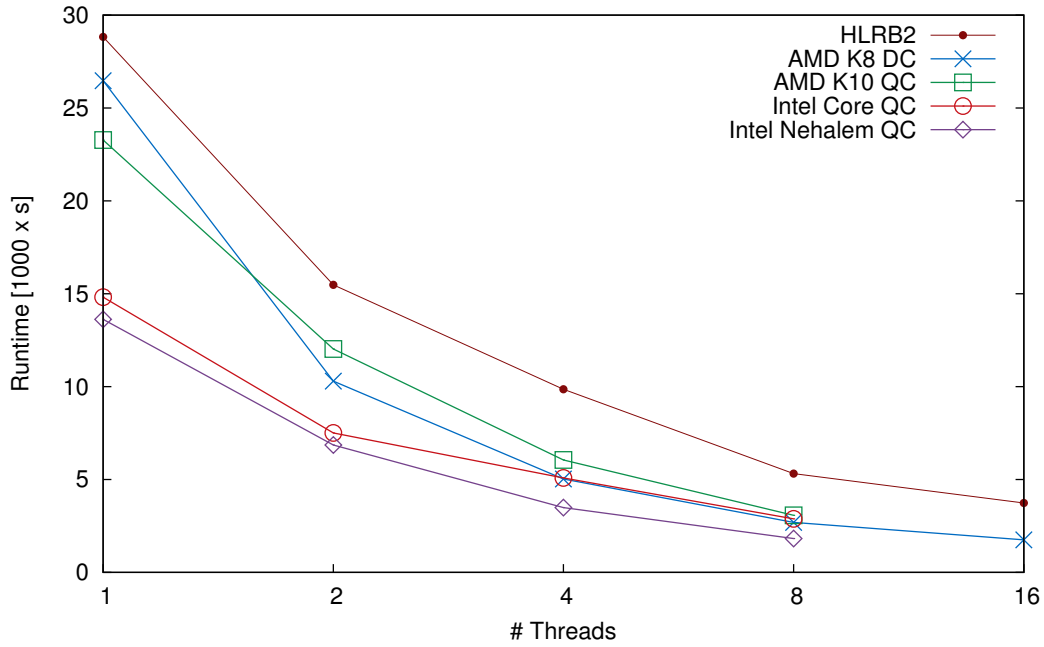

Figure 6.7: Execution times of the PThreads parallelization on the d500_5000 dataset.

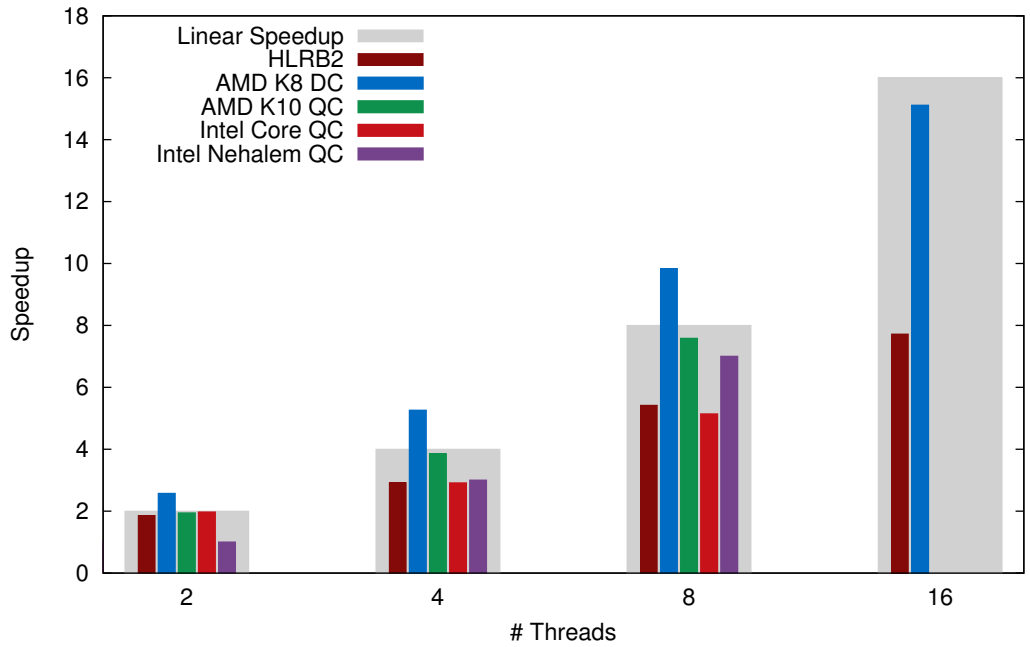

Figure 6.8: Speedups of the PThreads parallelization on the on d500_5000 dataset.

Dataset	#	K8		K10		Core		Nehalem		HLRB2	
		Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp
d50_50000	1	5295		5335		3382		2533		5801	
	2	2402	2.20	2670	2.00	1629	2.08	1219	2.08	3074	1.89
	4	1219	4.34	1384	3.85	882	3.83	618	4.10	1506	3.85
	8	648	8.18	778	6.85	719	4.70	329	7.71	913	6.35
	16	309	17.13							369	15.71
d500_5000	1	26467		23278		14815		13623		28828	
	2	10297	2.57	12027	1.94	7508	1.97	6857	1.99	15481	1.86
	4	5030	5.26	6054	3.85	5083	2.91	3486	3.91	9856	2.92
	8	2691	9.83	3072	7.58	2881	5.14	1822	7.48	5314	5.42
	16	1752	15.11							3736	7.72

Table 6.3: Runtimes and speedups (columns Time and SpUp) of the PThreads parallelization for 1-16 threads (column #) on the K8, K10, Core, Nehalem and HLRB2 systems.

respective results of the experiments with the OpenMP parallelization of `RAxML`. That is, the AMD platforms K8 and K10 as well as the Nehalem platform show a significantly better scaling behavior than the HLRB2 and the Core system. The most notable difference is the fact, that the PThreads parallelization yields considerably better execution times than the OpenMP parallelization with higher thread counts. This is especially true for the execution times of the *d500_5000* dataset on the HLRB2 that are up to 55% lower with the PThreads version. This is due to the fact that multiple jobs for computing the conditional probability vectors can be merged into a single job in the PThreads implementation which reduces the number of synchronization events. The experiments with the OpenMP implementation showed that the DSM architecture of the HLRB2 is very sensitive to the synchronization frequency. Consequently, it benefits greatly from this optimization. Because of its less favorable computation/communication ratio, this effect is more pronounced for the *d500_5000* dataset. The K8 system shows a runtime improvement of 33% for this dataset with 16 threads which results in an almost linear speedup of 15.11. The respective speedup of the OpenMP version was 10.11 for the same configuration.

Unfortunately, the PThreads parallelization approach does not always yield better runtimes. Especially in case of lower thread counts they are up to 28% higher than the respective runtimes of the OpenMP parallelization. Evidently, synchronization operations are the more expensive the more threads are involved. Therefore the reduction of the number of synchronization events in the PThreads parallelization is most beneficial in case of high thread counts. The poor performance in case of low thread counts can be explained by the poor performance of the synchronization operation the PThreads version uses. As the PThreads library provides no means for thread synchronization, a custom function had to be implemented for that purpose. This custom implementation utilizes a busy-wait approach that does not take the system's hardware topology into account. Obviously, this approach is inferior to the implementation that is provided by Intel's OpenMP compiler and requires further optimization. An optimized synchronization function would certainly improve the performance of the PThreads paralleliza-

tion not only for low thread counts but for all configurations. However, given the steadily increasing number of cores in current and upcoming microprocessors, good performance for high thread counts is of paramount importance for yielding reasonable runtimes on those processors. In addition to that, a slight performance disadvantage of the PThreads parallelization might be acceptable as it can be co-developed with MPI parallelization with little additional effort, whereas the OpenMP parallelization needs to be developed separately.

6.4 MPI

The performance and scalability of the fine-grained MPI parallelization of `RAxML` has been assessed on the clusters (TUM-IC and ICE) and supercomputers (BGL and HLRB2) listed in Section 6.1 with three different datasets: *d50_50000*, *d50_500000*, and *d250_500000* (see Table 6.1). Absolute runtimes and speedups of these experiments are reported in Table 6.4. Note that the speedups are based on the number of worker processes, i.e., the master process is disregarded since it does not contribute to the actual computations. Due to memory shortage and/or runtime restrictions no reference execution times with a single worker process could be measured for the *d50_500000* dataset on BGL and for *d250_500000* on neither system. In those cases, the reported speedups are relative to the run with the least feasible amount of workers.

The runtimes and speedups for the *d50_50000* dataset are depicted in Figure 6.9 and 6.10, respectively. All systems scale linearly up to 7 worker processes. For HLRB2 and the ICE system, the speedup degrades for more than 7 workers, whereas BGL scales linearly up to 31 worker processes and TUM-IC even shows super-linear speedups up to 31 workers. With 63 workers BGL yields a parallel efficiency of 90% which is still acceptable although it already starts to degrade at this point. The better scalability of the TUM-IC and the BGL system can be explained by their better communication to computation ratio due to their low-latency networks coupled with moderate per CPU computing power. This is especially true for the BlueGene/L system. The performance on the HLRB2 is comparable to the PThreads version, although the absolute execution times as well as the speedups tend to be slightly higher. At first sight, the poor scaling behavior of the ICE system with 15 or more workers suggests its inferiority to the TUM-IC and BGL system. However, given that 15 worker processes on the ICE system yield similar runtimes as 31 workers on TUM-IC and 63 workers on BGL, its performance can be considered to be balanced.

As can be seen from Figures 6.11 and 6.12, the global picture for the *d50_500000* dataset is comparable to *d50_50000*. However, due to the bigger problem size of this dataset, i.e., the number of distinct patterns as well as the memory footprint are approximately one order of magnitude larger, all systems scale to higher processor counts. With regard to scalability, the BGL outperforms all other systems as it scales almost linearly up to 127 workers. The speedups on HLRB2 and TUM-IC are comparable and start to degrade for more than 15 workers. However, they are is still acceptable up to 63 worker processes (parallel efficiency: 95% and 92%). The ICE system consistently scales sub-linearly but approaches the speedups of HLRB2 and TUM-IC for higher processor counts. With regard to absolute execution times,

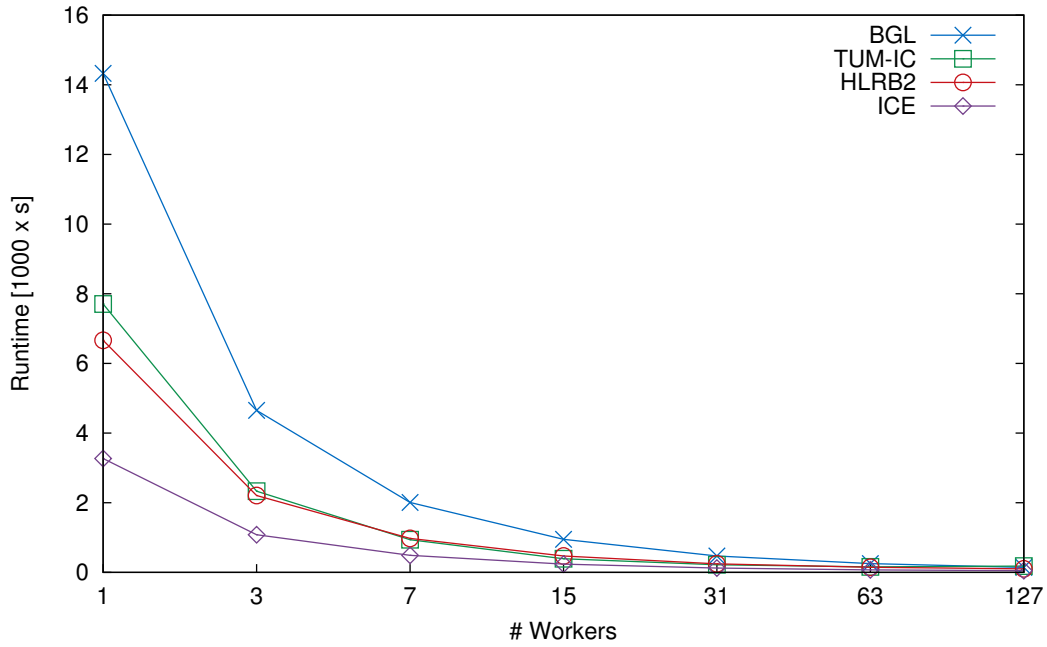

Figure 6.9: Execution times of the MPI parallelization on the d50_50000 dataset.

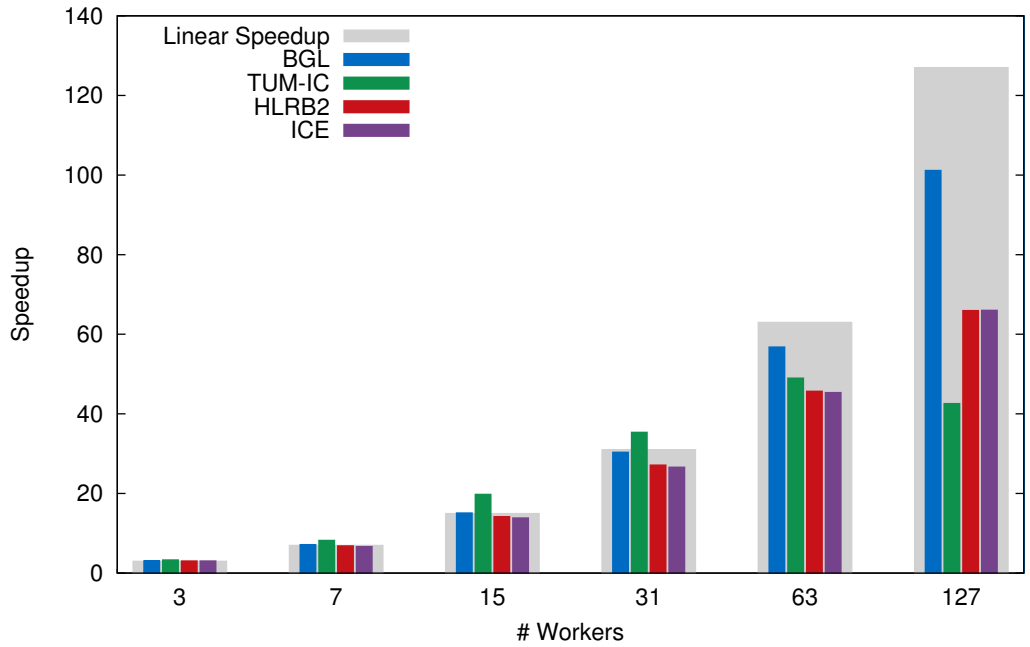

Figure 6.10: Speedups of the MPI parallelization on the d50_50000 dataset.

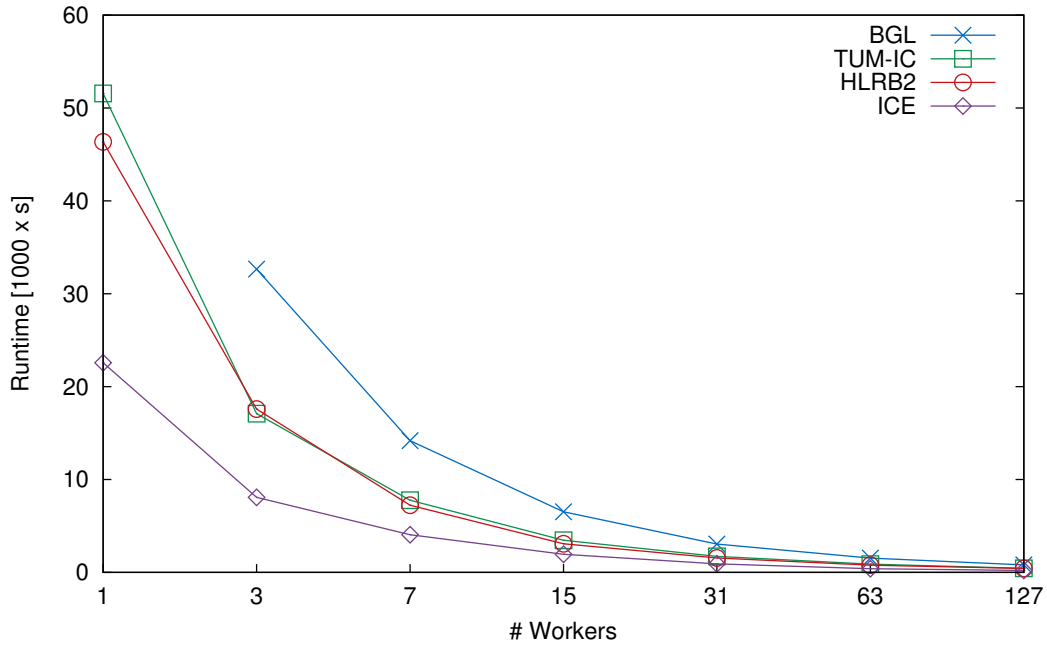

Figure 6.11: Execution times of the MPI parallelization on the MPI d50_500000 dataset.

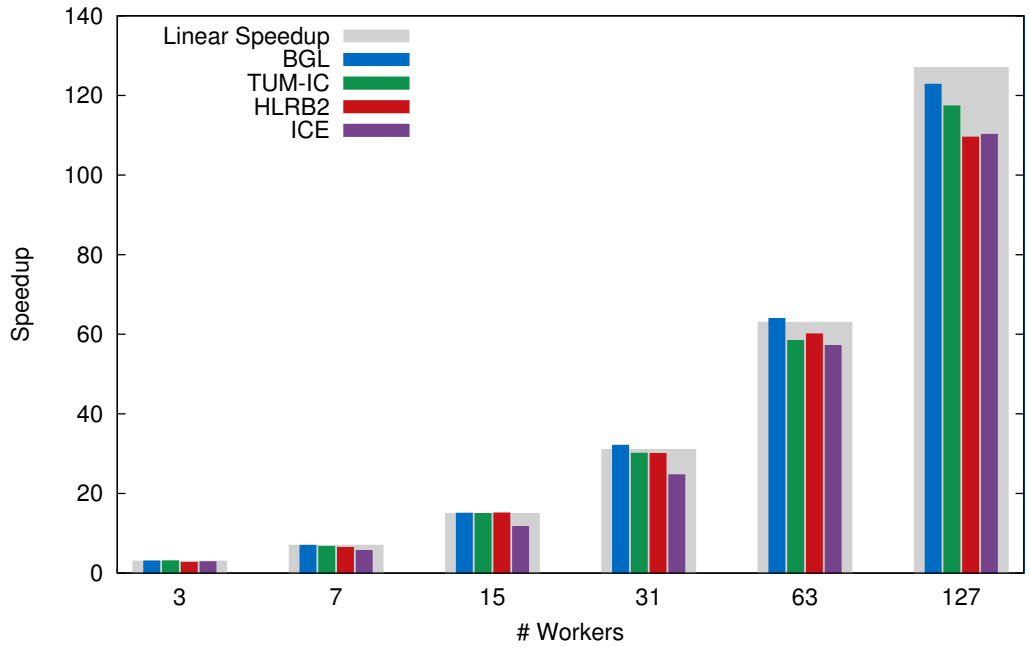

Figure 6.12: Speedups of the MPI parallelization on the d50_500000 dataset.

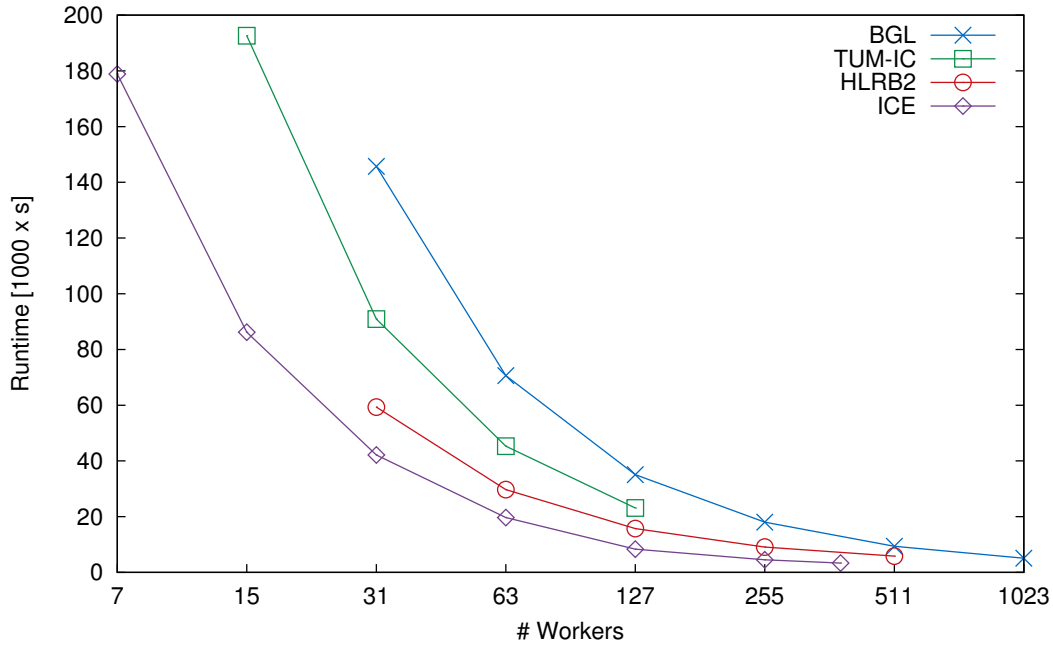

Figure 6.13: Execution times of the MPI parallelization on the d250_500000 dataset.

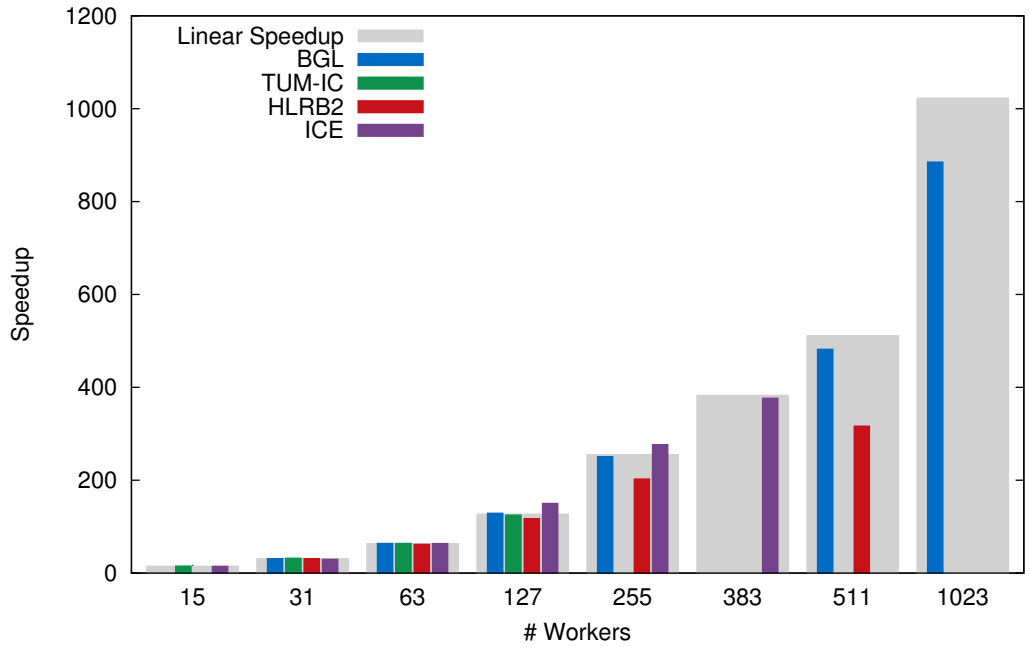

Figure 6.14: Speedups of the MPI parallelization on the d250_500000 dataset.

Dataset	Workers	BGL		TUM-IC		HLRB2		ICE	
		Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp	Time[s]	SpUp
d50_50000	1	14326		7705		6663		3271	
	3	4653	3.08	2332	3.30	2205	3.02	1080	3.03
	7	2008	7.14	937	8.22	975	6.83	490	6.67
	15	948	15.11	390	19.76	471	14.15	236	13.84
	31	472	30.36	218	35.36	246	27.12	123	26.58
	63	252	56.79	157	48.97	146	45.69	72	45.34
	127	142	101.16	181	42.59	101	65.94	50	66.02
d50_500000	1			51569		46353		22577	
	3	32659		17084	3.02	17589	2.64	8073	2.80
	7	14187	6.91	7766	6.64	7228	6.41	4049	5.58
	15	6531	15.00	3462	14.89	3075	15.07	1947	11.59
	31	3055	32.07	1716	30.04	1548	29.95	916	24.64
	63	1533	63.92	884	58.35	772	60.04	396	57.08
	127	798	122.75	440	117.33	423	109.48	205	110.17
d250_500000	7							178850	
	15			192591				86226	14.52
	31	145739		90901	31.78	59340		42140	29.71
	63	70617	63.98	45307	63.76	29716	61.90	19697	63.56
	127	35056	128.88	23111	125.00	15725	116.98	8346	150.00
	255	18024	250.65			9079	202.62	4525	276.66
	383							3324	376.69
	511	9375	481.93			5816	316.30		
	1023	5105	885.05						

Table 6.4: Runtimes and speedups (columns Time and SpUp) of the MPI parallelization for 1-1023 worker processes (column Workers) on the BGL, TUM-IC, HLRB2, and ICE system.

it outperforms all other systems by at least a factor of two.

Figures 6.13 and 6.14 depict the runtimes and speedups for the *d250_500000* dataset. Compared to *d50_500000*, the memory footprint of this dataset is by an order of magnitude larger. Due to the higher number of distinct sequence patterns, the conditional probability vectors are twice as long, which results in an improved computation/communication ratio. At the same time this ratio decreases because of the increased number of sequences in the alignment that results into an increased number of nodes and branches in the tree. However, this effect can be compensated up to a certain degree by the algorithmic improvements in the MPI version that merge consecutive ML-operations into a single job. Consequently, all systems scale up to higher processor counts for this dataset. The BGL scales super-linearly up to 127 worker processes and still shows a parallel efficiency of 94% for 511 workers and of 87% for 1023 workers. TUM-IC scales linearly up to its maximum number of 128 processors. The ICE system even shows super-linear speedups up to 255 worker processes and still yields a parallel efficiency of 98% at its maximum number of 384 cores. Although the HLRB2 shows only sub-linear speedups in all experiments, its relative runtime performance to the other systems increased significantly for this dataset: for the smaller 50-sequences datasets, the runtimes of the HLRB2 were almost identical to TUM-IC and more than twice as high as the runtimes of the ICE system. Yet, in case of the *d250_500000* dataset, the execution times of HLRB2 are approximately 35% lower than the corresponding times on TUM-IC and only 40% higher than on the ICE system. Ob-

viously, the bigger caches of the Itanium processor compared to the x86 architectures impact performance on big datasets.

6.5 Performance of the Hybrid MPI/MPI Parallelization

The performance of the hybrid MPI/MPI parallelization has been assessed using the *d50_50000* dataset. The experiments were conducted on BGL partitions of 32, 128, and 512 processors. For the sake of completeness, Tables 6.5 and 6.6 provide the absolute execution times of these experiments.

# Total processors	# Processors/Group				
	8	16	32	64	128
32	1,984s	963s			
128	1,986s	964s	502s	291s	
512	1,977s	963s	502s	291s	190s

Table 6.5: Runtimes for a single tree search on the d50_50000 dataset in multiple-groups configuration on the BGL system.

# Total Processors	# Processors/Group	
	16	32
32	15,387s	15,008s
128	3,850s	4,006s
512	963s	1,005s

Table 6.6: Runtimes for 32 distinct tree searches on the d50_50000 dataset in multiple-groups configuration on the BGL system.

Figure 6.15 shows execution times for individual tree inferences using groups of 8, 16, 32, 64, and 128 processors. The blue line shows the times for a single master-worker group (see Table 6.4). The remaining three graphs depict execution times for multiple master-worker groups on the aforementioned BGL partitions which have been split into 4, 8, 16, 32, and 64 groups (where applicable) using `MPI_Comm_split()` as described in Section 5.3.3. So for example, the red graph shows the runtimes of the experiments on a total number of 128 processors. The datapoint at "32 Processors/Group" depicts the average runtime of 4 groups with 32 processors each. Note that the experiments have been organized such that each group uses the same starting tree and hence yields identical results.

As expected, the runtimes observed for the multiple groups setup are slightly higher than the corresponding runtimes of a single master-worker group. This is due to the fact that on the BlueGene/L system messages are sent over the higher latency peer-to-peer network in case of the multi-group setups while a single group can utilize the faster specialized collective network (see Section 5.3.3). Note that on any other supercomputer or cluster that does not provide such a specialized collective network but utilizes a single network for all message types, no runtime differences should occur between the single-group and a multi-group setups.

Figure 6.16 depicts the total execution times for 32 distinct tree searches on 32, 128, and 512 processors that have been split into groups of 16 and 32 processors. So, for example, in case of 128 processors and 16 processors per group, 8 masters with their private set of 15 workers perform 32 distinct tree searches in parallel, four for each group. Again, identical starting trees have been used for the individual tree searches such that each experiment yielded 32 identical result trees. The plot shows that the total execution time decreases linearly with an increasing number of total processors used for computation. Furthermore, one can see that groups of 16

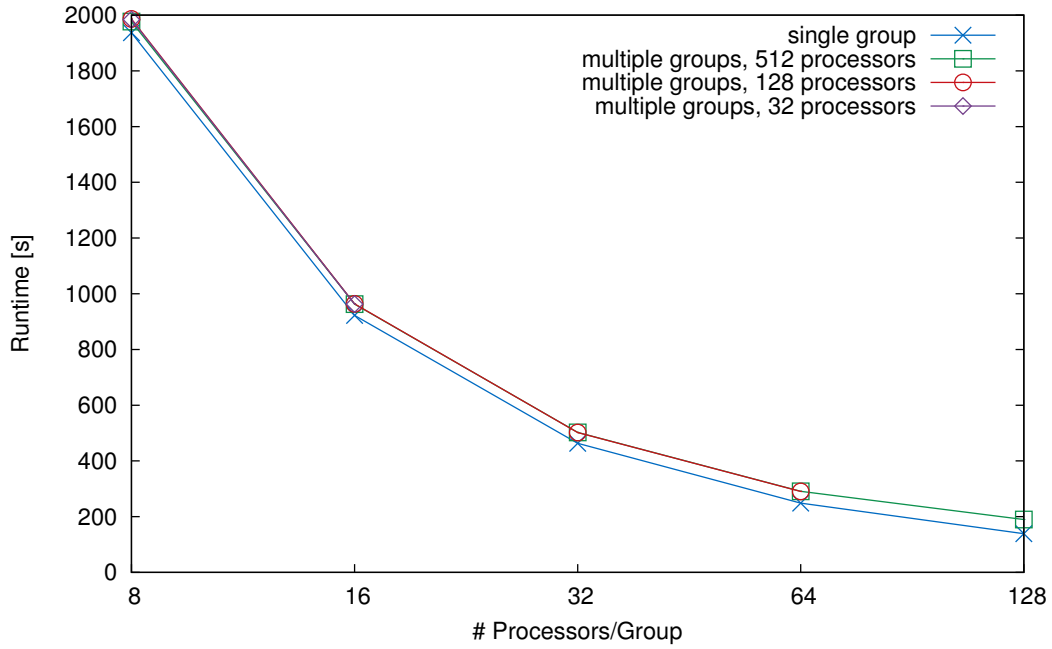

Figure 6.15: Execution times of multiple groups setup on the d50_50000 dataset.

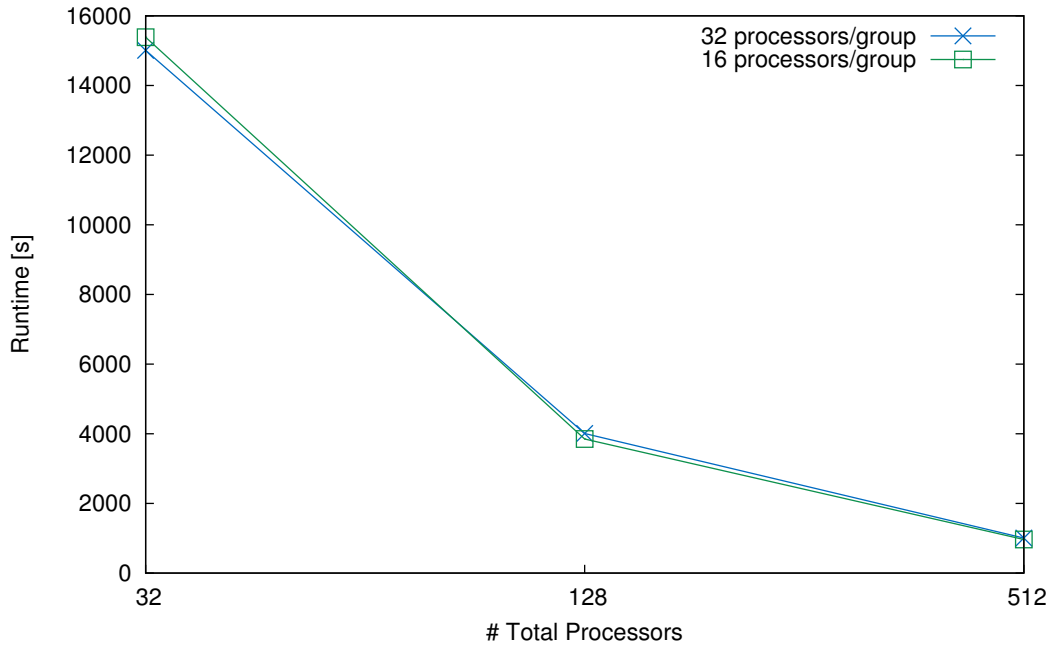

Figure 6.16: Execution times for 32 distinct tree searches on the d50_50000 dataset.

processors perform slightly better than groups of 32 processors – as expected, given the absolute execution times for single groups in Table 6.4.

The experiments show that the total number of distinct groups does not influence the runtime of the individual tree searches. That is, the runtime per tree search depends solely on the number of processes (workers) per group, not on the number of groups. This also proves that communication between sub-masters and the super-master is infrequent enough to not influence the fine-grained parallelism within each group.

Chapter 7

Conclusion and Future Work

This final chapter provides the conclusion of the work and gives an outlook on potential directions for future research.

7.1 Conclusion

The inference of large phylogenetic trees from molecular sequence data is still one of the grand challenges in bioinformatics. The steadily growing amount of sequence data that is available for phylogenetic analyses continuously poses new challenges for the developers of tree inference tools. As the input datasets for phylogenetic analyses are growing larger and larger, problems arise not only from the computational requirements of such analyses but also from their memory requirements. In fact, the memory requirements can easily exceed 100 GB and hence render a tree inference impossible on most workstations. Even on large HPC systems, that amount of memory is typically not available to a single process as they are composed from many smaller nodes with significantly less memory. Applications that aim at facilitating the inference of huge phylogenetic trees therefore have to meet two requirements: firstly, they need to distribute their computations over multiple processors in order to gain the required computational power. Secondly, they also need to distribute their data structures over multiple nodes for gathering the necessary amount of aggregate memory.

Inferring phylogenetic trees under the Maximum Likelihood criterion exhibits several potential sources of parallelism at different levels of granularity: loop-level parallelism, inference-level parallelism, and embarrassing parallelism. Although all of them can be exploited for distributing the computational workload, only the exploitation of the loop-level parallelism within the phylogenetic likelihood function allows for also distributing the data. Yet, exploiting such a fine-grained source of parallelism under a rather coarse-grained distributed memory programming paradigm like MPI – which is a necessity for acquiring huge amounts of aggregate memory – is not as straight forward as its exploitation under shared memory programming paradigms like OpenMP.

The need for parallel tools for phylogenetic analyses is not only driven by the molecular data flood, but also by the developments in computer architecture over recent years. Due to

the power wall, the performance of new computer systems can only be increased by increasing the degree of parallelism. This applies to computers of all scale – desktop computers as well as supercomputers. In the multi-core era, even the smallest notebooks have at least two cores, powerful workstations with less than sixteen cores have become rare, and supercomputers comprise thousands of cores. Consequently, parallel tools are required even for comparatively small phylogenetic analyses and large-scale analyses require highly optimized tools that are able to scale up to thousands of processors.

Apart from the specific problems in phylogenetic inference, the new parallel computer architectures pose new challenges for all application domains. As the multi-core trend continues, microprocessors will become increasingly complex and heterogeneous, and the total number of cores will increase in computers of all scales. As a high degree of complexity and parallelism is hardly manageable manually by the user, new automatic tools are required that allow for exploiting new architectures efficiently – for end-users as well as for developers.

The work described in this thesis mainly focuses on parallelization concepts for phylogenetic tree inference under the Maximum Likelihood criterion. The concepts have been implemented in `RAxML` under the shared memory programming paradgim (OpenMP and PThreads) as well as under the distributed memory programming paradigm (MPI). However, as the concepts are generic, they could easily be adopted in any other ML-based tool. Due to similarities of Bayesian and ML-based phylogenetic tree inference, most of the concepts can also be applied to Bayesian programs. The performance of the parallel implementations has been assessed on eight different computer systems with four different datasets. All implementations show good speedup values and in some cases even super-linear speedups. The MPI parallelization overcomes memory requirements as a limiting factor for phylogenetic analyses and has proven to scale up to 2048 processors. It has recently been chosen to become part of the SPEC MPI2007 benchmark suite which is used to assess the performance of large-scale computer systems. Furthermore, it has been used to perform the most computationally intensive real-world phylogenetic analysis to date for which over 2.25 million CPU hours have been utilized.

Besides the parallelization concepts for phylogenetic tree inference, the work also covered problems that are due to the continuously increasing degree of parallelism of new computer architectures: the novel properties of recent multi-core architectures have been analyzed and their potential performance bottlenecks described. Additionally, the `autopin` tool has been developed that allows for performance-optimized and automated thread pinning on these architectures and hence helps to exploit their performance efficiently by by-passing bottlenecks.

Additionally, new scalable approaches for identifying performance bottlenecks in MPI applications have been developed and implemented in the peformance analysis tool `Periscope`. They are based solely on summary information and therefore facilitate automatic performance analysis, even on large-scale distributed memory systems.

7.2 Future Work

The execution times and memory requirements for phylogenetic tree inferences depend mainly on the size of input sequence alignment. Although the parallelization concepts and their actual implementations described in this thesis allow for utilizing thousands of processors simultaneously and to overcome memory requirements as a limitation for phylogenetic analyses, they are only suitable for input datasets of a certain shape. This is due to the fact, that the loop-level parallelism within the phylogenetic likelihood functions scales only with the length of the conditional probability vectors of the tree nodes. As their length corresponds to the length of the sequences in the input alignment, this parallelization is especially useful for datasets with long sequences. In fact, there is no limit to the length of the sequences as long as enough processors and aggregated main memory are available for the tree inference.

However, the execution times and the memory requirements depend on both dimensions of the input data matrix: the length of the sequences as well as their number. Therefore, parallelization concepts need to be devised that also scale with the number of sequences. As indicated in Section 5.2.2, an additional source of parallelism can also be found at the inference level within the search algorithm that is used for inferring phylogenetic trees. Since input alignments with a large number of sequences yield tree topologies with a large number of nodes, it is likely that the search algorithm could operate on multiple independent subtrees in parallel. As described earlier, this source of parallelism would probably yield only a modest degree of parallelism due to hard to resolve data dependencies and therefore has been ignored so far. However, it could be combined with loop-level parallelism and would probably increase the combined degree of parallelism by one order of magnitude.

The memory requirements and execution times of phylogenetic analyses could additionally be reduced by handling gaps in the input sequence alignment properly. Due to the molecular data flood, analyses of so-called phylogenomic datasets that comprise hundreds or even thousands of genes have become very popular. However, these datasets tend to be very gappy as they contain species with disparate genes. In order to align the sequences properly, gaps need to be inserted for genes that do not exist in the sequence of the respective species. Currently, all tools for phylogenetic tree inference handle gaps as if they actually contained data, i.e., they allocate memory and compute conditional probability vector entries for them. As phylogenomic datasets frequently contain up to 90% of gaps, it evidently makes no sense to waste CPU cycles and memory on data that does not exist. An initial proof-of-concept implementation [138] and the follow-up work [132] by Stamatakis and colleagues showed, that the memory requirements can be reduced proportionally to the gappyness of the dataset and the execution times by up to one order of magnitude. However, as this approach makes intensive use of dynamic pointer meshes, its integration into a production-level application like `RAxML` poses a major software engineering challenge due to the complexity of the code and the large variety of models of sequence evolution that are supported.

The `autopin` tool could be improved in several ways: at the moment the user needs profound

knowledge of the hardware infrastructure (e.g., how many cores are available on how many sockets, how many cores are on a chip, which cores share caches, etc.) in order to choose a reasonable set of pinnings. To make the tool easier to use for people with no background in computer architectures, a mechanism could be implemented that automatically detects the hardware infrastructure and selects appropriate schedule mappings to be analyzed.

In order to accommodate distinct program phases within an application, it might be worthwhile to combine `autopin` with the OpenMP runtime. As the particular parallel regions of an OpenMP application would probably match the different program phases, this approach would allow for automatically detecting different phases. Additionally, as the OpenMP runtime typically creates the threads dynamically at the beginning of a parallel region, the number of threads could be chosen individually for each region in order to yield the best performance.

With regard to `Periscope`, future work should cover the design and implementation of more search strategies and performance properties. Additionally, it might be useful for the analysis of MPI applications to not only search for the properties of a particular process, but to also take the relationships to other processes into account. For example, the fact that one process calls the receive operation before another process called the corresponding send does not necessarily mean that this is an "Early Receiver" event. If several receivers came in too early, it is rather a "Late Sender" event. Such properties could be found with a multi-step search strategy.

Bibliography

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

[2] OProfile System Profiler for Linux. http://oprofile.sourceforge.net.

[3] Top500 Website. http://www.top500.org.

[4] Advanced Micro Devices, Inc. *AMD Opteron Processor Product Data Sheet*. Number 23932. 2007. http://support.amd.com/us/Processor_TechDocs/23932.pdf.

[5] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual*. Number 24593. 2007.

[6] Advanced Micro Devices, Inc. *Family 10h AMD Opteron Processor Product Data Sheet*. Number 40036. 2008. http://support.amd.com/us/Processor_TechDocs/40036.pdf.

[7] 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.

[8] M. Bane and G. Riley. Automatic Overheads Profiler for OpenMP Codes. In *Proceedings of the Second Workshop on OpenMP (EWOMP 2000)*, Edinburgh, UK, 2000.

[9] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, , and E. W. Sayers. GenBank. *Nucl. Acids Res.*, 37:D26–D31, 2009.

[10] 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.

[11] O. R. P. Bininda-Emonds, M. Cardillo, K. E. Jones, R. D. E. MacPhee, R. M. D. Beck, R. Grenyer, S. A. Price, R. A. Vos, J. L. Gittleman, and A. Purvis. The delayed rise of present-day mammals. *Nature*, 446:507–512, 2007.

[12] F. Blagojevic, A. Stamatakis, C. D. Antonopoulos, and D. S. Nikolopoulos. RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, 2007.

[13] B. Boussau and M. Gouy. Efficient likelihood computations with nonreversible models of evolution. *Systematic Biology*, 55(5):756–768, 2006.

[14] R. P. Brent. *Algorithms for Minimization without Derivatives*, chapter 4. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[15] 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.

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

[17] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[18] M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *Bioinformatics*, 4:66, 2003.

[19] H. Brunst and B. Mohr. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with VampirNG. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, 2005.

[20] D. Bryant. A classification of consensus methods for phylogenies. In M. F. Janowitz, F.-J. Lapointe, F. R. McMorris, B. Mirkin, and F. Roberts, editors, *Bioconsensus*, volume 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–184, 2003.

[21] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[22] 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.

[23] L. L. Cavalli-Sforza and A. W. F. Edwards. Phylogenetic analysis. models and estimation procedures. *American Journal of Human Genetics*, 19(3 Pt 1):233–257, 1967.

[24] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. *Lecture Notes in Computer Science*, 3746:415–425, 2005.

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

[26] G. Ciaccio and G. Chiola. GAMMA and MPI/GAMMA on Gigabit Ethernet. *Lecture Notes in Computer Science*, 1908(4):129–136, 2000.

[27] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, sep 1989.

[28] C. Darwin. *The Origin of Species*. John Murray, London, 1859.

[29] 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.

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

[31] 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.

[32] B. Efron. Bootstrap methods: Another look at the jackknife. *Annals of Statistics*, 7(1):1–26, 1979.

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

[34] G. Em Karniadakis and R. M. Kirby II. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2003.

[35] 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.

[36] S. Eranian. The perfmon2 Interface Specification. Technical Report HPL-2004-200R1, Hewlett-Packard Laboratory, February 2005.

[37] T. Fahringer, M. Gerndt, G. Riley, and J. Träff. Formalizing openmp performance properties with the apart specification language (asl). In *Proceedings of the International Workshop on OpenMP: Experiences and Implementation*, volume 1940 of *Lecture Notes in Computer Science*, pages 428–439. Springer, 2000.

[38] T. Fahringer, M. Gerndt, G. Riley, and J. Träff. Specification of performance problems in MPI-programs with ASL. In *Proceedings of the International Conference on Parallel Processing (ICPP'00)*, pages 51–58, 2000.

[39] T. Fahringer, M. Gerndt, G. Riley, and J. Träff. Knowledge specification for automatic performance analysis. Technical report, APART Working Group, 2001.

[40] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. The APART specification language. In *Proceedings of the 8th Workshop on Compilers for Parallel Computers (CPC'2000)*, pages 1–11, 2000.

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

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

[43] J. Felsenstein. Confidence limits on phylogenies: An approach using the bootstrap. *Evolution*, 39(4):783–791, 1985.

[44] 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.

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

[46] R. Fletcher. *Practical Methods of Optimization*. Wiley, 2000.

[47] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, 1994.

[48] K. Fürlinger and S. Moore. Continuous runtime profiling of openmp applications. In *Proceedings of the 2007 Conference on Parallel Computing (PARCO 2007)*, pages 677–686, 2007.

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

[50] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006)*, pages 303–312, Bonn, Germany, 2006.

[51] GenProbe. New oligonucleotides corresponding to HIV-1 sequences used for selective amplification and as hybridisation probes for detection of HIV-1, 1993.

[52] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[53] 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.

[54] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1982.

[55] N. Goldman and Z. Yang. A codon-based model of nucleotide substitution for protein-coding dna sequences. *Molecular Biology and Evolution*, 11(5):725–736, 1994.

[56] P. A. Goloboff, S. A. Catalano, J. M. Mirande, C. A. Szumik, J. S. Arias, M. Källersjö, and J. S. Farris. Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups. *Cladistics*, 25(3):211–230, 2009.

[57] 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.

[58] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[59] S. W. Graham, R. G. Olmstead, and S. C. H. Barrett. Rooting phylogenetic trees with distant outgroups: A case study from the commelinoid monocots. *Molecular Biology and Evolution*, 19:1769–1781, 2002.

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

[61] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference Volume 2 – The MPI-2 Extensions*. MIT Press, 1998.

[62] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[63] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 – Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[64] 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.

[65] J. Handy. *The Cache Memory Book*. Academic Press, 1998.

[66] 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.

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

[68] A. Hejnol, M. Obst, A. Stamatakis, M. Ott, G. W. Rouse, G. D. Edgecombe, P. Martinez, J. Baguñà, X. Bailly, U. Jondelius, M. Wiens, W. E. G. Müller, E. Seaver, W. C.

Wheeler, M. Q. Martindale, G. Giribet, and C. W. Dunn. Assessing the root of bilaterian animals with scalable phylogenomic methods. *Proceedings of the Royal Society B: Biological Sciences*, 276(1677):4261–4270, 2009.

[69] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, fourth edition, 2006.

[70] Hewlett-Packard/Intel/Microsoft/Phoenix/Toshiba. *Advanced Configuration and Power Interface Specification*, 2010.

[71] D. M. Hillis and J. J. Bull. An empirical test of bootstrapping as a method for assessing confidence in phylogenetic analysis. *Systematic Biology*, 42(2):182–192, 1993.

[72] J. P. Huelsenbeck and D. M. Hillis. Success of phylogenetic methods in the four taxon case. *Systematic Biology*, 42:247–264, 1993.

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

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

[75] M. Höhl, T. Rigoutsos, and M. A. Ragan. Pattern-based phylogenetic distance estimation and tree reconstruction. *Evol. Bioinf. Online*, 2:357–373, 2006.

[76] Intel Corporation. *Intel 64 and IA-32 Architectures: Software Developer's Manual*. Denver, CO, USA, 2007.

[77] Intel Corporation. *Intel 7300 Chipset Memory Controller Hub (MCH) Datasheet*. Number 318082. 2007. http://www.intel.com/Assets/PDF/datasheet/318082.pdf.

[78] Intel Corporation. *Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process Datasheet*. Number 30922106. 2007. http://download.intel.com/support/processors/sb/30922106.pdf.

[79] Intel Corporation. *Intel Pentium 4 Processor Supporting Hyper-Threading Technology*. Number 303128. 2007. http://download.intel.com/design/Pentium4/datashts/303128.pdf.

[80] Intel Corporation. *Quad-Core Intel Xeon Processor 3200 Series Datasheet*. Number 316133. 2007. http://www.intel.com/Assets/en_US/PDF/datasheet/316133.pdf.

[81] Intel Corporation. *Intel Xeon Processor 7200 Series and 7300 Series Datasheet*. Number 318080. 2008. http://www.intel.com/Assets/en_US/PDF/datasheet/318080.pdf.

[82] Intel Corporation. *Intel Xeon Processor 5500 Series Datasheet*. Number 321321. 2009. http://www.intel.com/Assets/en_US/PDF/datasheet/321321.pdf.

[83] D. T. Jones, W. R. Taylor, and J. M. Thornton. The rapid generation of mutation data matrices from protein sequences. *Computer applications in the biosciences : CABIOS*, 8(3):275–282, 1992.

[84] 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.

[85] 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.

[86] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(4), 2008.

[87] D. E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, 1974.

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

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

[90] P. O. Lewis. A likelihood approach to estimating phylogeny from discrete morphological character data. *Sys Bio*, 50(6):913–925, Oct 2001.

[91] S. Li. *Phylogenetic tree construction using Markov chain Monte carlo*. PhD thesis, Ohio State University, 1996.

[92] 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.

[93] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, and J. A. Miller. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.

[94] A. Matsunaga and J. A. Fortes. On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:495–504, 2010.

[95] B. Mau. *Bayesian phylogenetic inference via Markov chain Monte carlo methods*. PhD thesis, University of Wisconsin, 1996.

[96] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.3*. 2008.

[97] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2.* 2009.

[98] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21:1087–1092, June 1953.

[99] Microsoft Corporation. *Distributed Component Object Model (DCOM) Remote Protocol Specification Rev. 10.0*, 2009.

[100] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[101] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. Mate: Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007.

[102] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder. Spec mpi2007—an application benchmark suite for parallel systems using mpi. *Concurr. Comput. : Pract. Exper.*, 22(2):191–205, 2010.

[103] S. V. Muse and B. S. Gaut. A likelihood approach for comparing synonymous and nonsynonymous nucleotide substitution rates, with application to the chloroplast genome. *Molecular Biology and Evolution*, 11(5):715–724, 1994.

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

[105] 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.

[106] G. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek. fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Computer Applications in the Biosciences (CABIOS)*, 10:41–48, 1994.

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

[108] M. Ott. PAxML@home: Specification and Development of a Globally Distributed Software Architecture for Computation of Phylogenetic Trees. Master's thesis, Technische Universität München, 2004. In German.

[109] M. Ott, T. Klug, J. Weidendorfer, and C. Trinitis. autopin - Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In *Proceedings of the 1st Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2008.

[110] M. Ott and A. Stamatakis. Preparing RAxML for the SPEC MPI Benchmark Suite. In S. Wagner, M. Steinmetz, A. Bode, and M. M. Müller, editors, *High Performance Computing in Science and Engineering*. Springer, Garching, 2010.

[111] 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.

[112] M. Ott, J. Zola, S. Aluru, and A. Stamatakis. Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L. In *Proceedings of the 19th IEEE/ACM Supercomputing Conference 2007*, 2007.

[113] M. Ott, J. Zola, S. Aluru, and A. Stamatakis. ParBaum: Large-scale Maximum Likelihood-based Phylogenetic Analysis. In S. Wagner, M. Steinmetz, A. Bode, and M. Brehm, editors, *High Performance Computing in Science and Engineering*. Springer, Garching, 2008.

[114] 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.

[115] N. D. Pattengale, M. Alipour, O. R. Bininda-Emonds, B. M. Moret, and A. Stamatakis. How many bootstrap replicates are necessary? In *"Proceedings of RECOMB 2009"*, 2009.

[116] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fourth edition, 2008.

[117] M. Pettersson. Perfctr: Linux performance monitoring counters driver. Retrieved Dec 2009 from http://user.it.uu.se/~mikpe/linux/perfctr.

[118] W. Pfeiffer and A. Stamatakis. Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code. In *Proceedings of HICOMB 2010 (in conjunction with IPDPS 2010)*, 2010.

[119] M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, 2003.

[120] B. Rannala and Z. Yang. Probability distribution of molecular evolutionary trees: a new method of phylogenetic inference. *Molecular Evolution*, 43:304–311, 1996.

[121] J. Ripplinger and J. Sullivan. Does choice in model selection affect maximum likelihood analysis? *Systematic Biology*, 57(1):76–85, 2008.

[122] M. Ronaghi, M. Uhlén, and P. Nyrén. DNA SEQUENCING: A Sequencing Method Based on Real-Time Pyrophosphate. *Science*, 281(5375):363–365, 1998.

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

[124] M. S. Rosenberg and S. Kumar. Incomplete taxon sampling is not a problem for phylogenetic inference. *Proceedings of the National Academy of Sciences of the United States of America*, 98(19):10751–10756, 2001.

[125] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, M. van Waveren, , and B. Whitney. Large System Performance of SPEC OMP2001 Benchmarks. In *Proceedings of WOMPEI2002, the Workshop on OpenMP: Experiences and Implementations*, Kansai Science City, Japan, 2002.

[126] 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.

[127] M. J. Sanderson. Confidence Limits on Phylogenies: The Bootstrap Revisited . *Cladistics*, 5(2):113–129, 1989.

[128] L. T. Schermerhorn. Automatic Page Migration for Linux - A Matter of Hygiene, January 2007. Talk at linux.conf.au 2007.

[129] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

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

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

[132] A. Stamatakis and N. Alachiotis. Time and memory efficient likelihood-based tree searches on gappy phylogenomic alignments. *Bioinformatics*, 26(12):i132–i139, 2010.

[133] A. Stamatakis, P. Hoover, and J. Rougemont. A Fast Bootstrapping Algorithm for the RAxML Web-Servers. *Systematic Biology*, 57(5):758–771, 2008.

[134] A. Stamatakis, M. Lindermeier, M. Ott, T. Ludwig, and H. Meier. DAxML: A Program for Distributed Computation of Phylogenetic Trees Based on Load Managed CORBA. In *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT 2003)*, volume 2763 of *Lecture Notes in Computer Science*, pages 538–548. Springer, 2003.

[135] A. Stamatakis, T. Ludwig, and H. Meier. A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC2004)*, pages 197–201, 2004.

[136] A. Stamatakis, T. Ludwig, and H. Meier. RAxML-II: A Program for Sequential, Parallel & Distributed Inference of Large Phylogenetic Trees. *Concurrency and Computation: Practice and Experience (CCPE)*, 17:1705–1723, 2005.

[137] 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.

[138] 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.

[139] A. Stamatakis and M. Ott. Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study. In M. Chetty, A. Ngom, and S. Ahmad, editors, *Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008)*, volume 5265 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 2008.

[140] A. Stamatakis and M. Ott. Load Balance in the Phylogenetic Likelihood Kernel. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*, 2009.

[141] A. Stamatakis, M. Ott, and T. Ludwig. RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *Lecture Notes in Computer Science*, 3606:288–302, 2005.

[142] Sun Microsystems Inc. *Java RMI Specification Rev. 1.10*, 2004. Retrieved Jan 2009 from http://java.sun.com/j2se/1.5.0/docs/guide/rmi.

[143] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):758–771, 2005.

[144] D. L. Swofford. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4.* Sinauer Associates, Sunderland, Massachusetts, 2003.

[145] D. L. Swofford, G. J. Olsen, P. J. Wadell, and D. M. Hillis. Phylogenetic inference. In D. M. Hillis, C. Moritz, and B. K. Mabel, editors, *Molecular Systematics*, pages 407–514. Sinauer Associates, Sunderland, MA, 1996.

[146] The International HapMap Consortium. The International HapMap Project. *Nature*, 426:789–796, 2003.

[147] 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.

[148] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Traff. Investigating High Performance RMA Interfaces for the MPI-3 Standard. In *Proceedings of the International Conference on Parallel Processing (ICPP'09)*, pages 293–300. IEEE Computer Society, 2009.

[149] J. Treibig and G. Hager. likwid - A tool collection for multi threaded high performance programming. Retrieved Dec 2009 from http://code.google.com/p/likwid.

[150] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14:46–55, 1997.

[151] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Computational Biology*, 1(4):337–348, 1994.

[152] J. Weidendorfer, M. Ott, T. Klug, and C. Trinitis. Latencies of Conflicting Writes on Contemporary Multicore Architectures. In *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT 2007)*, volume 4671 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2007.

[153] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison Wesley, second edition, 1994.

[154] 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.

[155] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, Feb. 2003.

[156] The World Wide Web Consortium (W3C). *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, 2007.

[157] Z. Yang. Maximum likelihood estimation of phylogeny from DNA sequences when substitution rates differ over sites. *Molecular Biology and Evolution*, 10:1396–1401, 1993.

[158] Z. Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. *Molecular Evolution*, 39(3):306–314, 1994.

[159] Z. Yang. Among-site rate variation and its impact on phylogenetic analyses. *Trends in Ecology & Evolution*, 11(9):367–372, 1996.

[160] Z. Yang. Maximum Likelihood Estimation on Large Phylogenies and Analysis of Adaptive Evolution in Human Influenza Virus A. *Molecular Evolution*, 51:51–423, 2000.

[161] Z. Yang. *Computational Molecular Evolution.* Oxford University Press, 2006.

[162] L. Zaslavsky and T. A. Tatusova. Accelerating the neighbor-joining algorithm using the adaptive bucket data structure. *Lecture Notes in Computer Science*, 4983:122–133, 2008.

[163] D. Zwickl. *Genetic Algorithm Approaches for the Phylogenetic Analysis of Large Biological Sequence Datasets under the Maximum Likelihood Criterion.* PhD thesis, University of Texas at Austin, April 2006.

[164] D. J. Zwickl and D. M. Hillis. Increased taxon sampling greatly reduces phylogenetic error. *Systematic Biology*, 51(4):588–598, 2003.