Scalable Evolutionary Bioinformatics

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation
von

Klaus Lukas Hübner

Tag der mündlichen Prüfung: 30. Juni 2025

1. Referent:	Prof. Dr. Alexandros Stamatakis Foundation for Research and Technology – Hellas Heidelberger Institut für Theoretische Studien Karlsruher Institut für Technologie
2. Referent:	Prof. Dr. Bertil Schmidt Universität Mainz

Trees sprout up just about everywhere in computer science.

TAoCP, Vol. IV-A
Donald E. Knuth

Abstract

The research field of evolutionary bioinformatics describes the evolutionary origins of organisms, primarily using (mathematical) trees. These trees are based on statistical models that account for variations among the observed genomic data. While *phylogenetic* trees describe the shared evolutionary history among distinct species, *genealogical* trees consider distinct individuals of the same species. The amount of available genomic data grows exponentially, and therefore presents major challenges for applications in evolutionary bioinformatics. For instance, as single-core performance plateaus, we require both, work-efficient and highly-parallel algorithms. However, existing petascale systems experience hardware failures 0.8 to 5.7 times per day, and future exascale systems are expected to fail every 30 to 60 minutes. To avoid loosing progress, algorithms should mitigate from these failures. Further, algorithms that yield bit-identical results across different hardware environments strengthen scientific claims. Finally, software libraries facilitate efficient application development by providing abstractions to manage the growing complexity of scientific codes.

In this thesis, we address four concrete instances of the challenges outlined above: We address the *reproducibility challenge* by presenting the first parallel phylogenetic tree inference tool that yields bit-identical results under varying core counts. To this end, we also present a bit-reproducible operation-agnostic distributed reduction algorithm. Additionally, we address the *algorithmic challenge* by proposing to compress sets of genealogical trees into a Directed Acyclic Graph and thus encode shared subtrees only once. This enables straightforward memoization, thereby yielding substantial runtime reductions over the state-of-the-art. Moreover, we address the *hardware failure challenge* by substantially accelerating the failure recovery of a fault-tolerant RAxML-NG version – a widely-used tool for phylogenetic tree inference. To this end, we present the, to the best of our knowledge, first general-purpose checkpointing solution allowing for in-memory recovery after a fail-stop hardware-failure without requiring spare cores (shrinking recovery). Finally, we address the *software complexity challenge* by developing high-level C++ bindings for MPI. Our C++ bindings enable a faster, less error-prone development of distributed applications, including access to MPI's failure-mitigation mechanisms and our bit-reproducible parallel reduce implementation.

Reproducibility Challenge Maximum-Likelihood phylogenetic tree inference searches for the tree and evolutionary model that best explain the observed genomic data. The likelihood score calculations at different genomic loci/sites are independent, and thus they are commonly computed in parallel and then reduced into a single tree score. However, fundamental arithmetic operations on IEEE 754 floating-point numbers, such as addition, inherently introduce rounding errors, causing the order of floating-point operations to affect the result. Consequently, how common parallel reduction algorithms re-associate operations under varying core counts, results in different round-off errors. We observe this effect to cause phylogenetic tree searches to diverge under varying degrees of parallelism for 31 % of 10 130 empirical datasets. 8 % of these divergent datasets even yield trees that are significantly worse than the best found tree (AU-test, $p < 0.05$). To alleviate these diverging results, we present a variant of RAxML-NG that ensures bit-reproducible results under varying core-counts, with a

slowdown of only 0 to 6.7 % (median 0.93 %) on up to 768 cores. To this end, we introduce the general-purpose distributed reduction algorithm ReproRed, which ensures bit-identical results under varying core counts by maintaining a fixed order of operations independent of the communication pattern. Thus, ReproRed is applicable to all associative reduction operations – in contrast to competitors, which are confined to summation only. ReproRed exchanges only the theoretical minimum number of messages, overlaps communication with computation, and utilizes a fast base-case for conducting local reduction. Further, ReproRed is able to all-reduce (via a subsequent broadcast) 4.1×10^6 operands across 48 to 768 cores in 19.7 to 48.61 µs, exhibiting a slowdown of 13 to 93 % over a non-reproducible all-reduce algorithm and 8 to 25 % over an non-reproducible Reduce-Bcast algorithm. ReproRed outperforms the state-of-the-art reproducible algorithm (summation only) beyond 10 000 elements per core.

Algorithmic Challenge In population genetic studies of recombining organisms (e.g., humans), sets of genealogical trees provide a more comprehensive view on evolution than a single tree, as commonly used in phylogenetics. Here, each of the (highly-correlated) trees describes the evolutionary history of a distinct genomic region. The current state-of-the-art data structure compresses these trees via edit operations when moving from one tree to the next along the genome. In contrast, we propose to compress the input trees into a Directed Acyclic Graph, such that subtrees are encoded only once, even if they appear multiple times in the input. This allows for a straight-forward memoization of intermediate results. Queries on our data structure are 2.1 to 11.2 (median 4.0) times faster than the state-of-the-art for obtaining important population genetic statistics such as Patterson's f, Tajima's D, pairwise Lowest Common Ancestor, and others on empirical, and simulated datasets.

Hardware-Failure Challenge When mitigating hardware failures, it is often impractical to either request replacement for failed cores, or to maintain spares. Thus, algorithms must continue with the remaining cores, necessitating them to restore lost data and to redistribute the workload and respective data to the remaining cores. However, current checkpointing libraries often do not support such a data redistribution. Moreover, they typically write checkpoints to the parallel file system instead of the main memory, resulting in slow recoveries due to low disk access speeds and disk/network congestion. We present ReStore, which, in turn, supports the data redistribution that is required by shrinking recovery and backs-up the application data in memory. ReStore reloads lost data in milliseconds on up to 24 576 cores and thereby accelerates the reloading of data during failure recovery for the failure-tolerant version of RAxML-NG by one to two orders of magnitude.

Software Complexity Challenge The Message Passing Interface (MPI) abstracts over message exchanges in parallel distributed-memory machines. However, although high-level programming languages such as C++ yield software development faster and less error-prone, MPI does not provide C++ bindings. To this end, we propose such bindings along an extensively tested, open-source implementation, KaMPIng, that covers all abstraction levels from low-level MPI calls to convenient STL-style bindings. Through a configurable inference of parameter defaults, fine-grained memory allocation control, enhanced safety guarantees, and a flexible plugin system, KaMPIng enables both, rapid prototyping and careful fine-tuning of distributed algorithms. By utilizing template-metaprogramming, KaMPIng incurs (near) zero running time overhead. We also integrate ReproRed as well as abstractions over MPI's failure-mitigation interface into KaMPIng and showcase KaMPIng via multiple application benchmarks. Various scientific C++ projects already utilize KaMPIng and we

are working with the MPI language binding working group towards the development of novel MPI C++ bindings with KaMPIng as a reference implementation.

Impact Our work allows researchers to analyze the exponentially growing phylogenetic and population genetic datasets, to perform novel, work-intensive, analyses, publish reproducible results, and to develop complex distributed algorithms through appropriate abstractions. We demonstrate this by developing the, to the best of our knowledge, first bit-reproducible phylogenetic tree inference tool as well as by substantially accelerating recovery of the fault-tolerant RAxML-NG version. Thus, our contributions push the scalability frontier in evolutionary bioinformatics in particular and High Performance Computing in general.

Zusammenfassung

Das Forschungsfeld der Evolutionären Bioinformatik beschreibt die Evolutionsgeschichte von Organismen, primär mithilfe von (mathematischen) Bäumen. Diese Bäume basieren auf statistischen Modellen, welche die Differenzen zwischen den beobachteten Genomen erklären. Hierbei beschreiben *Phylogenetische Bäume* die gemeinsame Evolutionsgeschichte zwischen Organismen verschiedener Spezies und *Genealogische Bäume* die Evolutionsgeschichte zwischen Individuen derselben Spezies. Die Menge der verfügbaren genomischen Daten wächst exponentiell und stellt damit Anwendungen der Evolutionären Bioinformatik vor komplexe Herausforderungen. Zum Beispiel stagniert die Rechengeschwindigkeit einzelner Rechenkerne, was sowohl arbeitseffiziente als auch hoch-parallele Algorithmen notwendig macht. Allerdings beobachten wir schon bei heutigen Petascale-Systemen pro Tag 0.8 bis 5.7 Hardwareausfälle und für zukünftige Exascale-Systeme werden solche Ausfälle alle 30 bis 60 Minuten erwartet. Damit eine Anwendung bei einem Hardwareausfall nicht ihren kompletten Fortschritt verliert, muss sie mit diesen umgehen können. Des Weiteren fördern Algorithmen, welche bitidentische Ergebnisse über verschiedenste Hardwareumgebungen ausgeben, die Glaubwürdigkeit der von ihnen erzeugten wissenschaftlichen Ergebnisse. Schlussendlich sind Softwarebibliotheken unabdingbar für die effiziente Softwareentwicklung, da sie Abstraktionen bereitstellen, um mit der wachsenden Komplexität wissenschaftlicher Programme umzugehen.

In dieser Thesis befassen wir uns mit vier konkreten Instanzen der oben beschriebenen Herausforderungen: Wir gehen die *Reproduzierbarkeits-Herausforderung* an, indem wir das erste parallele, bitreproduzierbare Programm für Phylogenetische Inferenz vorstellen. Hierfür stellen wir einen operatorunabhängigen, verteilten Reduktionsalgorithmus vor, welcher bitidentische Ergebnisse unabhängig von der Anzahl der Rechenkerne garantiert. Wir gehen die *Algorithmische-Herausforderung* an, indem wir eine neue Datenstruktur vorschlagen, welche eine Menge genealogischer Bäume zu einem Directed Acyclic Graph komprimiert und dabei geteilte Teilbäume nur einmal speichert. Dies erlaubt die einfache Wiederverwendung von Zwischenergebnissen und reduziert dadurch die Laufzeit gegenüber dem bisherigen Stand der Wissenschaft substantiell. Des Weiteren gehen wir die *Hardwareausfall-Herausforderung* an, indem wir die, nach unserem besten Wissen, erste allgemein anwendbare Checkpointing-Bibliothek präsentieren, welche das Wiederherstellen des Programmzustands nach einem fail-stop Hardwareausfall aus dem Arbeitsspeicher erlaubt, ohne dabei zusätzliche Rechenknoten zu benötigen. Mithilfe dieser Checkpointing-Bibliothek erreichen wir eine substantielle Beschleunigung der ausfalltoleranten RAxML-NG Version – ein weit verbreitetes Programm zur Inferenz Phylogenetischer Bäume. Schlussendlich gehen wir die *Software-Komplexitäts-Herausforderung* an, indem wir eine hoch-abstrahierte C++-Schnittstelle für MPI entwickeln, welche eine schnellere, weniger fehlerbehaftete Entwicklung von verteilten Anwendungen ermöglicht. Sie erlaubt zudem den Zugriff auf die Fehlermitigationsmechanismen von MPI 5, auf unseren bitreproduzierbaren parallelen Reduktionsalgorithmus, sowie weitere Abstraktionen.

Reproduzierbarkeits-Herausforderung Die Maximum-Likelihood basierte Inferenz Phylogenetischer Bäume sucht denjenigen Baum und dasjenige evolutionäre Modell, welche die beobachteten genomischen Daten am besten erklären. Die Likelihood-Werte verschiedener

genomischer Loki/Sites sind unabhängig voneinander und werden daher oft parallel berechnet und anschließend reduziert, um einen einzelnen Baumwert zu erhalten. Jedoch sind grundlegende arithmetische Operationen auf IEEE 754 Fließkommazahlen, wie z. B. Addition, inhärent mit Rundungsfehlern behaftet. Dies führt dazu, dass die Reihenfolge, in welcher Rechenoperationen ausgeführt werden, den Wert des Endergebnisses beeinflussen. Übliche verteilte Reduktionsalgorithmen assoziieren die Reduktionsoperationen je nach Anzahl der Rechenkerne anders, was wiederum in unterschiedlichen Rundungsfehlern resultiert. Wir beobachten, dass dieser Effekt Phylogenetische Baumsuchen, welche auf unterschiedlich vielen Rechenknoten durchgeführt wurden in 31 % von 10 130 empirischen Datensätzen divergieren lässt. Auf 8 % der divergierenden Datensätze werden sogar Bäume gefunden, welche signifikant schlechter sind als der beste gefundene Baum (AU-Test, $p < 0.05$). Als Lösungsansatz entwerfen wir eine Variante von RAxML-NG, welche bitreproduzierbare Ergebnisse unabhängig von der Anzahl der Rechenkernen garantiert. Unsere bitreproduzierbare Variante ist nur 0 bis 6.7 % (Median 0.93 %) langsamer als die nicht-reproduzierbare Referenz auf bis zu 768 Rechenkernen. Hierfür stellen wir einen allgemein-anwendbaren und verteilten Reduktionsalgorithmus namens ReproRed vor, welcher die Reihenfolge der Rechenoperationen unabhängig vom Kommunikationsmuster konstant hält. Damit erreicht ReproRed, dass Reduktionen über Variationen in der Anzahl Rechenkerne hinweg bitidentische Ergebnisse liefern und auf alle assoziativen Reduktionsoperationen anwendbar sind – im Gegensatz zu Mitbewerbern, welche nur Summierung unterstützen. ReproRed tauscht nur das theoretische Minimum an Nachrichten aus, überlappt Kommunikation mit Berechnung und setzt einen schnellen Basisfall für lokale Reduktionen ein. ReproRed kann (mithilfe eines anschließenden Broadcast) 4.1×10^6 Operanden, welche über 48 bis 768 Rechenkerne verteilt sind, in 19.7 bis 48.61 µs reduzieren. Damit ist ReproRed nur 13 bis 93 % langsamer als ein nicht-reproduzierbarer Allreduce-Algorithmus und nur 8 bis 25 % langsamer als ein nicht-reproduzierbarer Reduce-Bcast Algorithmus. ReproRed ist schneller als der bisher schnellste reproduzierbare Algorithmus (unterstützt nur Summierung) auf mehr als 10 000 Elementen pro Rechenknoten.

Algorithmische-Herausforderung In populationsgenetischen Untersuchungen auf rekombinierenden Organismen (z. B. Menschen) liefert die Verwendung von Mengen genealogischer Bäume ein umfassenderes Bild der Evolution als die Verwendung eines einzelnen Baumes, wie es in der Phylogenetik üblich ist. Hierbei beschreibt jeder dieser (hoch korrelierten) Bäume die Evolutionshistorie einer spezifischen genomischen Region. Die bisher schnellste Datenstruktur komprimiert diese Menge an Bäumen mithilfe von Edit-Operationen zwischen entlang des Genoms adjazenten Bäumen. Im Gegensatz dazu schlagen wir eine Codierung vor, in welcher Teilbäume nur einmal gespeichert werden, auch wenn sie mehrmals in den Eingabebäumen vorkommen. Dies erlaubt die einfache Wiederverwendung von Zwischenergebnissen. Statistische Anfragen an unsere Datenstruktur sind 2.1 bis 11.2 (Median 4.0) mal schneller als an die bisher schnellste bekannte Datenstruktur. Dies umfasst die Berechnung von wichtigen populationsgenetischen Statistiken, wie z. B. Pattersons f, Tahimas D und paarweisen Lowest Common Ancestor-Anfragen, auf empirischen und simulierten Datensätzen.

Hardwareausfall-Herausforderung Bei der Behandlung von Hardwareausfällen ist es für Programme sowohl impraktikabel Ersatz für die ausgefallenen Rechenkerne anzufragen, als auch sich diese von Programmstart an in Reserve zu halten. Daher ist es notwendig, dass Algorithmen die verlorenen Anwendungsdaten auf den verbleibenden Rechenkernen wiederherstellen und die verbleibende Arbeit auf ebendiese umverteilen. Allerdings unterstützen existierende

Checkpointing-Bibliotheken die dafür notwendige Datenumverteilung nicht. Zudem schreiben diese ihre Sicherungskopien oft auf das parallele Dateisystem anstatt in den Hauptspeicher der Rechenknoten, was in langsamen und konkurrierenden Zugriffen auf die entsprechenden Festplatten resultiert. Wir stellen ReStore vor, eine Checkpointing-Bibliothek, welche im Vergleich zu Mitbewerbern die für diese Wiederherstellung notwendige Umverteilung der Daten unterstützt und die Anwendungsdaten in den Hauptspeicher sichert. In Experimenten mit bis zu 24 576 Rechenkernen konnte ReStore durch einen Hardwareausfall verlorene Daten in Millisekunden wiederherstellen und beschleunigt damit die Fehlerbehandlung in der fehlertoleranten Version von RAxML-NG um ein bis zwei Größenordnungen.

Software-Komplexitäts-Herausforderung Das Message Passing Interface (MPI) abstrahiert über den Nachrichtenaustausch in Maschinen mit verteiltem Speicher. Obwohl hochabstrakte Programmiersprachen wie C++ die Softwareentwicklung schneller und weniger fehleranfällig gestalten, umfasst MPI keine C++-Schnittstelle. Wir entwerfen daher eine ebensolche und stellen mit KaMPIng eine quelloffene und ausgiebig getestete Implementierung zur Verfügung. KaMPIng deckt alle Abstraktionsebenen von unmittelbaren MPI-Aufrufen bis hin zu STL-ähnlichen Funktionsschnittstellen ab. Durch eine konfigurierbare Inferenz von Standardparametern, feingranularer Speicherallokationskontrolle, verbesserten Sicherheitsgarantien und ein flexibles Pluginsystem ermöglicht KaMPIng sowohl die schnelle Entwicklung von Prototypen als auch die Feineinstellung verteilter Algorithmen. Durch den Einsatz von Template-Metaprogrammierung erreicht KaMPIng all dies (fast) ohne zusätzliche Laufzeitkosten. Wir demonstrieren die Fähigkeiten von KaMPIng mithilfe mehrerer Beispielanwendungen und integrieren einen bitreproduzierbaren Reduktionsalgorithmus sowie eine Abstraktion der MPI-Fehlertoleranzmechanismen in KaMPIng. Verschiedene wissenschaftliche C++-Projekte verwenden bereits KaMPIng und wir arbeiten derzeit zusammen mit der MPI-Arbeitsgruppe für Programmiersprachenschnittstellen an der Entwicklung neuer C++-Schnittstellen für MPI – mit KaMPIng als Referenzimplementierung.

Bedeutung Unsere Arbeit erlaubt Wissenschaftler:innen die Analyse exponentiell wachsender Phylo- und Populationsgenetischer Datenmengen, neue, arbeitsintensive Analysen, die Publikation reproduzierbarer Ergebnisse und die Entwicklung komplexer, verteilter Algorithmen durch geeignete Abstraktionen. Wir demonstrieren dies durch die Entwicklung des, nach unserem besten Wissen, ersten bitreproduzierbaren Programms zur Phylogenetischen Bauminferenz sowie der substantiellen Beschleunigung der Hardwareausfall-Mitigation in der fehlertoleranten Version von RAxML-NG. Unser wissenschaftlicher Beitrag verschiebt damit die Grenze der Skalierbarkeit in der evolutionären Bioinformatik im Speziellen und Höchstleistungsrechnen im Allgemeinen.

Acknowledgements

I would like to thank my supervisors, Prof. Dr. Alexandros Stamatakis and Prof. Dr. Peter Sanders, who enabled me to employ the tools of algorithm engineering to exciting real-world challenges. I am thankful for their valuable feedback, opening new investigative directions, and providing guidance while allowing me the freedom to choose which paths to pursue and emphasize. Ultimately, my PhD often felt like I got the best of both research groups, for which I am extremely thankful. I also extend my thanks to Prof. Dr. Bertil Schmidt for kindly agreeing to review this work.

Furthermore, my gratitude goes to my colleagues, collaborators, and friends at the Algorithm Engineering Group at KIT and the Exelixis Lab at HITS for the shared coffee, excellent tea, laughter, and conversations among like-minded nerds. I greatly appreciate them sharing their incredibly deep knowledge about tooling, their feedback on my projects, and them taking the time review parts of this work.

On a more personal note, I want to express my gratitude to Julia for her love, support, patience, and encouragement. I am deeply grateful for my friendship with the Radonerds: Max, Silas, Balázs, Annika, Steffi, and Lena; with special thanks to Max for our endless discussions about the PhD journey. I will miss our shared bicycle tours and conversations. I also thank my parents for their never-ending support: From giving me my first science books over twenty years ago to accepting laundry baskets full of those books during vacations as well as supporting me throughout my studies.

I also thank an anonymous reviewer for pointing out that the simple approximation of the irrecoverable data loss given in Section 4.4.4 is very accurate for small f.

Funding The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`). Part of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and Arts Baden-Württemberg and by the Federal Ministry of Education and Research. Part of this work was funded by the Klaus Tschira foundation. This work was supported by a grant from the Ministry of Science, Research and Arts of Baden-Württemberg (Az: 33-7533.-9-10/20/2) to Peter Sanders and Alexandros Stamatakis. This project has received funding from the European Research Council (ERC)[1] under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

[1] 

Table of Contents

1 Introduction

Attribution: I am the sole author of the *text* in this chapter. Attributions of the *work* referenced herein can be found in the respective chapters.

Charles Darwin famously wrote that all living organisms share a common evolutionary history, which can be modeled using a tree [Dar59; Dar87]. Since then, hand-drawn trees based on morphological features [Dar59; Hae66; Hit40] have been replaced by trees that are computed using mathematical and statistical models. These models take into account variations among the genomic data of the observed species [Hin+15; Pen04]. The field of evolutionary bioinformatics models and analyzes the evolutionary history of organisms, either between species (phylogenetic trees) or between distinct individuals of the same species (genealogy). For example, a tree depicting how contemporary species of the *Rodentia* order relate to each other constitutes a phylogenetic tree, while a tree depicting the evolutionary history of different *Homo sapiens* groups alive today constitutes a genealogical tree (Section 2.1).

Both, phylogenetic and genealogical models, advance our understanding of evolutionary processes such as mutations, genetic drift, gene flow, or natural selection [Lui+18] and help us infer the tree of life [Fox+80; WKW90]. Their downstream applications include, among others, medical research [All64; Cas+12; CG14; Oka+20; Som+17; WN91] for example during the recent SARS-CoV19 pandemic [Ste+20], forensics [APV18; Ber+07; Ou+92], and wildlife conservation [HFR20; Hug+23; Rol+11; SS18] (Section 2.1).

Conceptually, genealogical trees are a magnified version of phylogenetic trees, further resolving the evolutionary relationships inside a single tip (species) of the phylogenetic tree down to the level of individuals (Section 2.1). However, this invalidates an important simplification: Single trees cannot fully describe processes where individuals pass on different parts of their genome independently [Mor16]. This can be observed, for example, in sexually reproducing species with multiple genome copies (e.g., two in humans) [Hud83]: Here, a process called *recombination* causes the genetic material, which is organized in a DNA strand, to break and reconnect before being passed on to an offspring. Thus, the closer two genetic positions are located with each other along the genome, the higher the chance to have been inherited from the same ancestor becomes [SK06]. Therefore, for recombining organisms, modeling the (correlated) history of distinct regions along the genome using different (correlated) trees captures the evolutionary history more accurately than a single tree [Hud83] (Section 2.1).

1.1 Scalability Challenges in Evolutionary Bioinformatics

Next Generation Sequencing has boosted the rate at which new genomic data is generated. Molecular data continues to grow exponentially, doubling roughly every 18 months [NHS25; SRA25] (Figure 1.1). Notably, the growth of available genomic data outpaces that of computational power and storage capabilities per unit of money [BDY16; GMM16; Li+19; Sat+23; Sau+19; Sch+18; Ste+15], which doubles every 24 months according to Moore's Law [Moo75]. According to Oliva *et al.* [Oli+24], next to the decreasing sequencing cost, other factors that contribute to this exponential growth include large-scale population genomics

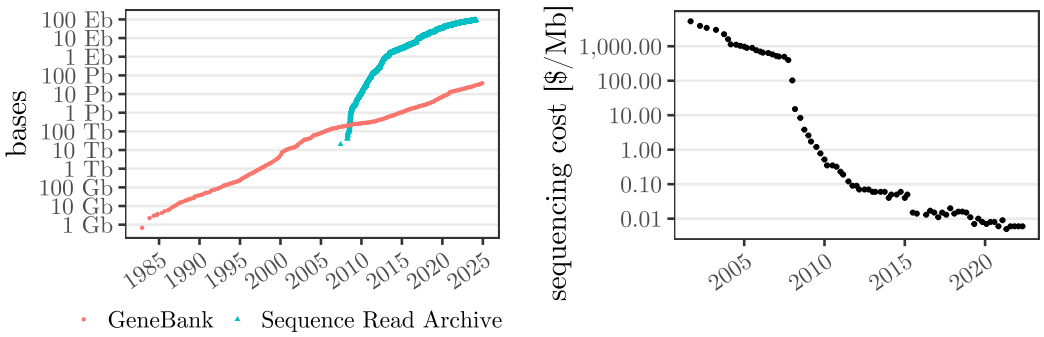


Figure 1.1 Data growth in evolutionary bioinformatics. Left: Amount of sequence data in the Sequence Read Archive [LSS10] and GenBank [Ben+12]. Data from the US National Library of Medicine [NHS25; SRA25]. Right: Average cost of sequencing per mega-base (Mb). Data from the US National Human Genome Research Institute [NHGRI25].

initiatives [Bic+24; Byc+18; FS13; Lei+14; Won+23a], genetic tests available directly to consumers [Sto+16], and efforts to integrate genome testing into public health care.

Today, nearly one hundred petabytes of raw sequencing data are publicly available, for example, in the Sequence Read Archive [LSS10; SRA25]. Half a million human genomes are available in the UK Biobank [Byc+18]. Analogous government-driven efforts are being undertaken, for example, in the USA [Bic+24], Estonia [Lei+14], Singapore [Won+23a], or the European Union [Sau+19]. Other collections – also containing up to hundreds of thousands of genomes – focus on representing the full diversity of human genetic material [Aut+15; Kar+20; Mal+16; Rei19; Tal+21]. Overall, Birney *et al.* [BVG17] estimate, that over 60 million people will have their genome sequenced in a healthcare context by the end of 2025. For non-human species, the Darwin Tree of Life project [Bla+22] aims to sequence 70 000 species from Ireland and Great Britain. Further initiatives focus on the genomes of plants [BdF24; Che+18], fungi [Gri+13], or eukaryotic species in general [Lew+18].

However, this exponentially growing amount of available data needs to be processed, necessitating efficient algorithms, implementations yielding bit-wise reproducible results, increased parallel scalability, and more complex software.

The Algorithmic Challenge As single processor speeds tend to stagnate [TW17], we can no longer rely on increasing sequential processing speeds to handle huge amounts of data. Instead, we require algorithms that perform the minimally required work by the problem[1] (that is, work-efficient algorithms), and highly optimized implementations, which exploit modern hardware features and leverage data locality. For example, the current state-of-the-art data structure used to compute population genetic statistics on the tree sets and associated sequences of genealogical datasets is `tskit` (Chapter 3). It compresses the sets of correlated genealogical trees via edit operations when moving from one tree to the next along the genome. This prevents `tskit` from re-using subtrees that are shared between non-adjacent trees along this tree sequence. As a solution, we propose a data structure, which compresses these genealogical trees by storing shared subtrees only once, thereby enabling straight-forward memoization of intermediate results, and yielding a more work-efficient algorithm as well as a substantial runtime reduction.

[1] For some computational problems, the minimum work required might not be known.

The Hardware Failure Mitigation Challenge Next to work-efficient data structures and algorithms, we also require increased parallelism to handle the data avalanche in evolutionary bioinformatics. However, with an increasing number of processors, the probability that a program experiences a hardware failure during its execution increases. Petascale systems already fail 0.8 to 5.7 times per day (Section 2.7). In fact, future exascale systems are expected to experience hardware failures every 30 to 60 minutes [Cap+14; DHR15; Sni+14]. Therefore, mitigating these hardware failures constitutes a major challenge for future exascale systems [SDM10]. During failure-recovery, algorithms have to be able to restore lost data the failed processors were working on. A common failure-mitigation strategy is to frequently write out the current state to a so-called checkpoint, which the program can subsequently reload after a failure to resume execution (Section 2.8). However, current checkpointing libraries typically write to the parallel file system (PFS), inducing slow recoveries due to low disk access speeds and disk congestion. Moreover, on modern cluster systems, it is impractical to request replacement for failed CPU cores, as well as to maintain spares. Thus, applications must continue working with the remaining cores, which requires redistributing the workload. However, many currently available checkpointing libraries support neither such a partial restore operation nor the respective data redistribution.

The Reproducibility Challenge Next to disseminating a result, scientific publications also serve to convince the reader of the correctness of this result [Mes10]. Computational reproducibility is considered a cornerstone for validating scientific claims [IT18] (Section 2.10). However, fundamental arithmetic operations such as additions or multiplications on IEEE 754 [IEEE754] floating-point numbers always induce rounding errors [Gol91]. Among other factors (Section 2.10), the mere number of CPU cores an application utilizes influences the order by which operations are executed and thus rounded [Li+23; SMR24]. Moreover, the impact of these low-level differences in floating-point values on high-level results of scientific codes have been reported for various fields [Cle+13; Die12; DN15; HD01; RRA11; Tau+10; Vil+09; Wie+19], including phylogenetics [DFS18; She+20a]. Additionally, in failure-mitigating algorithms, a hardware failure changes the number of CPU cores a program is executed on at unpredictable points during its runtime (Section 2.7).

Maximum-Likelihood based phylogenetic inference tools like RAxML-NG search for the phylogenetic tree and evolutionary model that best explain the observed genomic data (Section 2.3). Common parallel implementations compute the likelihood scores of the genomic sites in parallel and then sum over them in order to obtain an overall score for the tree. However, as parallel reduction algorithms that are highly common in scientific codes reassociate operations depending on the number of CPU cores used, this induces different round-off errors and therefore results, depending on the degree of parallelism (Section 2.10).

The Software Complexity Challenge Brooks [Bro87] describes software systems as being among the most complex human constructs. Requirements for efficiency, parallelism, and bit-reproducibility further increase their complexity. The fundamental tool for handling software complexity is abstraction, often in the form of functions that are bundled in software libraries. Thus, in order to facilitate the development of efficient, parallel, and bit-reproducible software, software libraries providing high-quality implementations are required. For example, the Message Passing Interface (MPI) [MPI4.1] offers abstractions over passing messages between processes in the context of High Performance Computing (HPC; Section 2.6). However, MPI only provides C and FORTRAN bindings, even though high-level programming languages such as C++ yield software development quicker and less error-prone.

Summary In conclusion, evolutionary bioinformatics faces challenges due to the exponential growth of genomic data caused by the advent of Next Generation Sequencing. To cope with this data avalanche and develop scalable codes we require work-efficient algorithms, programs that are able to mitigate hardware failures and thereby enable us to scale to more CPU cores, as well as techniques for handling additional sources of non-reproducibility introduced by parallelism. Moreover, we require abstractions in the form of software libraries to handle the increased software complexity associated with these changes. While all the challenges outlined in this section are particularly prominent in evolutionary bioinformatics because of the exponential data growth, they also apply to other areas of scientific computing.

1.2 Scientific Contributions

In the remainder of this work, we address the four major scalability challenges in evolutionary bioinformatics outlined in Section 1.1: We design, implement, and evaluate a work-efficient data structure for statistical queries on population genetics datasets, a novel scalable in-memory replicated storage for failure-tolerant applications, a distributed-memory reduction algorithm that guarantees reproducible results irrespective of the core-count, and high-level C++ bindings for MPI, enabling a faster, less error-prone development of distributed applications, including access to MPI's failure-mitigation mechanisms and our reproducible parallel reduction algorithm. While motivated by challenges in evolutionary bioinformatics, most of our results are also applicable to other domains of High Performance Computing. Our work allows researchers to analyze the exponentially growing phylogenetic and population genetic datasets, to perform novel, work-intensive, analyses, publish reproducible results and facilitates the development of complex distributed-memory algorithms through abstractions.

> *In **Chapter 3**, we address the **algorithmic challenge**, particularly the question: How can we design a data structure for sets of genealogical trees and associated genomic sequences, allowing for the efficient computation of statistical queries commonly executed on these population genetic datasets?*

We propose a novel data structure for storing sets of genealogical trees and their associated genomic sequences that allows for memoizing intermediate results. We show, that this enables devising work-efficient statistical queries and yields a substantial runtime reduction over the state-of-the-art data structure called *tree sequences*. Tree sequences compress sets of genealogical trees via edit operations when moving from one tree to the next along the genome. However, this hinders re-using intermediate results between non-adjacent trees in numerous cases. Our proposed data structure, genealogical forests, compresses the set of genealogical trees into a Directed Acyclic Graph (DAG; Chapter 3). In this DAG, identical subtrees that are shared across the input trees are encoded only once, thereby allowing for straight-forward memoization of all intermediate results. Additionally, we provide a C++ implementation of our proposed data structure, called `gfkit`. Our implementation is 2.1 to 11.2 (median 4.0) times faster at computing important population genetics statistics such as the Allele Frequency Spectrum, Patterson's f, the Fixation Index, Tajima's D, pairwise Lowest Common Ancestor (LCA), and others than the state-of-the-art tool on empirical and simulated datasets. On LCA queries with more than two samples as input, `gfkit` scales asymptotically better than the state-of-the-art implementation, and is therefore up to two orders of magnitude faster in our experiments. Our improvements will boost the development of novel analyses and models in the field of population genetics and increases scalability to the exponentially-growing genomic datasets.

*In **Chapter 4**, we address the **hardware failure challenge**, particularly the question: How can we design and implement a data distribution and recovery scheme for failure-tolerant algorithms that enables rapid recovery as the number of CPUs and thus the frequency of hardware failures increases?*

We advance the field of mitigating hardware failures in HPC by presenting an algorithmic framework and its C++ library implementation, ReStore, which enables MPI programs to perform scalable data-recovery when mitigating hardware failure (Chapter 4). By storing data in memory via an appropriate data distribution and replication scheme, recovery is substantially faster than for checkpointing approaches that rely on the PFS. Developers can specify which data to load. ReStore supports both, shrinking recovery and recovery via spare nodes. We evaluate ReStore in controlled, isolated environments and with real applications. Our experiments show millisecond loading times for lost input data on up to 24 576 cores. We also obtain a substantial recovery time speedup for the fault-tolerant version of the widely used phylogenetic tree search tool RAxML-NG. To the best of our knowledge, ReStore is the first in-memory checkpointing library that supports shrinking recovery. Therefore, it enables the development of algorithms that do not need to maintain spare CPU cores as replacement resources, and thereby avoid wasting computational resources.

*In **Chapter 5**, we address the **reproducibility challenge**, in particular the following two questions: How does the number of CPU cores used for a phylogenetic tree inference impact the inferred tree topology on empirical datasets, and how can we implement a phylogenetic tree inference that is numerically stable under varying core-counts? Further, how can we design a distributed-memory reduction algorithm that is agnostic of the used reduction operator but still yields bit-identical results when executed on a varying number of CPU cores?*

We provide a large-scale analysis of the bit-reproducibility of parallel phylogenetic inference algorithms under varying degree of parallelism. We find that these tree searches diverge on 31 % of 10 130 empirical datasets. More importantly, 8 % of these diverging searches result in trees that are significantly worse than the best found tree (AU-test, $p < 0.05$).

To alleviate this issue, we implement a variant of the widely-used phylogenetic tree inference tool RAxML-NG, which yields bit-reproducible results under varying core counts. This variant is merely 0 to 6.7 % (median 0.93 %) slower than non-reproducible RAxML-NG on up to 768 cores.

We also introduce ReproRed, a bit-reproducible reduction algorithm, which specifies the order of reduction operations independent of the communication pattern. Thus, ReproRed is applicable to arbitrary associative reduction operators – in contrast to its competitors, which are confined to summation. This allows us to integrate ReproRed into a open-source MPI wrapper library, which enables users to pass arbitrary reduction operators, such as C++ lambdas or function pointers. Furthermore, ReproRed exchanges only the theoretical minimum number of messages, overlaps communication with computation, and utilizes a fast base-case for conducting local reduction.

We find that ReproRed is able to all-reduce (via a subsequent broadcast) 4.1×10^6 operands on 48 to 768 cores in 19.7 to 48.61 µs. Thus, ReproRed exhibits a slowdown of 13 to 93 % over an non-reproducible all-reduce algorithm and 8 to 25 % over an non-reproducible Reduce-Bcast algorithm. ReproRed outperforms the state-of-the-art reproducible all-reduce algorithm ReproBLAS (summation only) when reducing more than 10 000 elements per core.

> *In **Chapter 6**, we address the **software complexity challenge**, particularly the question: How can we design and implement C++ language bindings for the Message Passing Interface (MPI), covering all abstraction levels, allowing for accelerated software development with (nearly) zero runtime overhead?*

We propose novel C++ language bindings that cover all abstraction levels from low-level MPI calls to convenient STL-style bindings with automatic inference of missing parameters where feasible. Further, we provide an open-source implementation of these bindings, called KaMPIng, which enables rapid prototyping as well as fine-tuning runtime behavior and memory management. A flexible type system and additional safeness guarantees contribute to preventing programming errors. Further, we employ template-metaprogramming to ensure that the compiler generates only those code paths that are required for computing the missing parameters for a specific function invocation. This results in (near) zero runtime overhead bindings. We demonstrate that KaMPIng provides a strong foundation for a future distributed standard library using multiple application benchmarks, from sorting algorithms to phylogenetic interference using RAxML-NG. Additionally, we re-implement our bit-reproducible distributed-memory reduction algorithm and integrate MPI's failure-mitigation bindings into KaMPIng.

Minor Contributions

My additional scientific contributions that do not form part of this thesis include, among others, work on accelerating the inference of tree sequences, porting phylogenetic likelihood-calculations to the GPU, investigating the numerical stability and hyperparameter optimization of phylogenetic inference, and developing a scalable fault-tolerant MapReduce implementation.

Re-Engineering Genomic Tree Sequence Inference Algorithms As already mentioned, we model the evolutionary history of recombining organisms via a set of genealogical trees, where each tree describes the local ancestries of a small genomic region (Section 2.1). When designing work-efficient statistical queries for these datasets in Chapter 3, we assume that these trees have already been inferred and are provided as input. Johannes Hengstler investigated algorithmic approaches for accelerating this inference in his Master's Thesis [Hen24], which I supervised. We propose several optimizations over the state-of-the-art tool `tsinfer`, improve cache efficiency, and reduce the number of CPU instructions. Further, we identify a limitation in `tsinfer`'s parallelization scheme and propose an appropriate novel alternative. We provide an open-source[2] Rust implementation of our improved algorithm and show that it outperforms the state-of-the-art by a factor of 1.9 to 2.4 for inferring ancestries on data from the Thousand Genomes Project. We also show that our parallelization scheme outperforms `tskit`'s from 32 cores upwards. We discussed the algorithmic improvements we achieved with our proof-of-concept implementation with the authors of `tskit`, who plan to adapt these new insights into `tsinfer`.

Acceleration of Phylogenetic Maximum-Likelihood Computations on GPU Phylogenetic inference (Section 2.3) on increasingly large datasets is computationally demanding with likelihood computations dominating the runtimes. I supervised Christoph Stelz' student

[2] `https://github.com/Cydhra/libairs`

project, where we present a novel GPU implementation of these likelihood calculations. It was designed for integration into the commonly used phylogenetic tree inference software RAxML-NG. We find, that for a full tree likelihood evaluation on large, unpartitioned datasets, our GPU implementation is 9 times faster than the highly-optimized, multithreaded CPU implementation of RAxML-NG (with site-repeats and patter-compression disabled!). Further, our implementation was 1.5 times slower than the corresponding BEAGLE GPU implementation of likelihood evaluation – an existing GPU-enabled and widely used likelihood computation library. However, during a phylogenetic tree search, the values of only 2 to 3 nodes in the phylogenetic tree typically need to be re-evaluated. When only re-evaluating the likelihood for a single tree node, both, our and BEAGLE's, GPU implementations are slower than RAxML-NG's CPU implementation for small to medium datasets and only slightly faster on large datasets (with more than 700 000 patterns[3]). We conclude that an efficient feature-complete GPU-parallelization requires a substantial amount of further work, including porting additional necessary functionality for phylogenetic tree inferences to the GPU. However, this might be unavoidable provided that most HPC systems that are currently under development rely on GPUs instead of CPUs for their bulk of computational power.

Numerical Stability of Phylogenetic Datasets At the start of the COVID-19 pandemic, I contributed to a project that analyses the stability of phylogenetic inference on the SARS-CoV19 genome data. My contribution consists in assessing the impact of the minimum branch length threshold on the likelihood of the inferred trees as well as evaluating the numerical stability of the parameter optimization for the free-rates [Sou+12; Yan95] and Γ-model [Yan94] of rate-heterogeneity. We found that the then-available SARS-CoV19 dataset exhibits a rugged likelihood surface [Sta10] (i.e., is numerically unstable), and therefore analyses on it and conclusion drawn from them are unreliable. We conclude, that respective publications on SARS-CoV19 phylogenies should be interpreted with extreme caution. Our work was published in Molecular Biology and Evolution [Mor+20a]. Julia Haag later generalized our findings to other datasets in her Master's Thesis [Haa21], which I supervised. She later developed them into the concept of "difficulty" for a phylogenetic dataset.[4]

Optimization of RAxML-NG's Hyperparameters Moreover, I contributed to Julia Haag's investigation of the impact of numerical thresholds on the time-to-convergence of numerical optimizations during phylogenetic tree searches. My contributions include supervising her Master's Thesis [Haa21] and modifying RAxML-NG in order to enable the user to explicitly set these numerical thresholds via command line parameters. By adjusting two of these thresholds, we accelerate RAxML-NG's phylogenetic tree inference mode by a factor of 1.9 ± 0.8 compared to the previous default settings, depending on the data collection. We published this work in Bioinformatics Advances [Haa+23], with me being a second author.

Scalable Failure-Tolerant MapReduce Together with Demian Hespe, I supervised Charel Mercatoris' Master's Thesis on developing and implementing a scalable failure-tolerant

[3] A pattern is a unique combination of genetic states across the genomic sequences.

[4] Note that I supervised Julia Haag's Master's Thesis but did not contribute to J. Haag et al. "From Easy to Hopeless – Predicting the Difficulty of Phylogenetic Analyses". In: *Molecular Biology and Evolution* 39.12 (2022). Edited by N. Saitou. ISSN: 1537-1719. DOI: 10.1093/molbev/msac254.

MapReduce framework for HPC systems. We observe that the full state of a MapReduce algorithm is described by its network communication and exploit this insight in order to create a full backup of the program's state with minimal additional communication. We published this work as a technical report on arXiv [Hes+24b], with me being a first author.

Coraxlib I also contributed to the development of a new version of the phylogenetic likelihood library formally called LibPLL-2 [Flo+14], now named Coraxlib. My main contribution is a complete re-write of the CMake code used to build the library. Further, minor, contributions include bug fixes and other maintainability enhancements such as adding unit-tests and missing parts of the documentation. Currently, Coraxlib is used by RootDigger [BS21], GeneRax [Mor+20c], AleRax [Mor+24], Asteroid [MWS22], Adaptive-RAxML-NG [Tog+23], and RAxML-NG v2 (unreleased).

1.3 Outline of this Thesis

The remainder of this thesis is structured as follows: First, we provide an overview of the concepts and terms used in this thesis (Chapter 2). We then design, implement, and evaluate an efficient data structure for computing population genetic statistics on genealogical trees and associated sequences and implement them in `gfkit` (Chapter 3). In Chapter 4, we discuss our ReStore C++ software library, which enables the scalable recovery of failure-mitigating, distributed algorithms. Next, we analyze the numerical reproducibility in phylogenetic inference algorithms and present our operation-agnostic distributed-memory reduction algorithm as well as a bit-reproducible version of the phylogenetic inference tool RAxML-NG (Chapter 5). Moreover, in Chapter 6, we introduce our (near) zero-overhead C++-abstractions for the Message Passing Interface and finally conclude in Chapter 7.

2 Background

Attribution: I am the sole author of this chapter. However, some formulations, sentences, and paragraphs have been copied either in modified or verbatim form from the following publication, where I am a (sometimes shared) first author:

1. L. Hübner and A. Stamatakis. "Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests". In: *24th Workshop on Algorithms in Bioinformatics (WABI)*. edited by S. P. Pissis and W. Sung. Volume 312. Leibniz International Proceedings in Informatics (LIPcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pages 1–22. ISBN: 978-3-95977-340-9. DOI: 10.4230/lipics.wabi.2024.5
2. L. Hübner et al. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *12th IEEE/ACM Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE Press, 2022, pages 24–35. DOI: 10.1109/ftxs56515.2022.00008
3. L. Hübner et al. "Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". in: *Bioinformatics* 37.22 (2021). Edited by R. Schwartz, pages 4056–4063. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab399
4. T. N. Uhl et al. "KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI". in: *High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pages 689–709. DOI: 10.1109/sc41406.2024.00050

Summary: We provide an overview of the concepts and terms used in the remainder of this thesis. First (Section 2.1), we introduce trees as a model of evolutionary relationships, both, for describing the shared evolutionary history between species (phylogenetics), as well as between individuals of the same species (genealogy). Next, we summarize how these trees are inferred on the basis of the differences in the genetic code of organisms (Section 2.2) using maximum-likelihood based phylogenetic inference (Section 2.3). Further, we introduce statistical measures that are commonly used in population genetics to analyze and describe genealogical trees (Section 2.4). Next, we describe the machine model we employ to analyze and asses our parallel algorithms and their parallel efficiency (Section 2.5) as well as the Message Passing Interface (MPI), which we use in our implementations (Section 2.6). In Section 2.7, we discuss hardware failures in High Performance Computing (HPC) systems, which represents one of the major challenges for upcoming exascale systems. We also discuss various strategies for mitigating these failures. We subsequently introduce a classification system for the most prevalent failure-handling strategy, checkpoint/restart (Section 2.8) and provide background information on mechanisms for mitigating failures in the upcoming MPI 5 standard (Section 2.9). We conclude with a discussion about the inherent challenges of devising bit-reproducible parallel scientific codes given the non-associativity of IEEE 754 floating-point operations (Section 2.10).

2.1 Trees as a Model of Evolution

In the mid 19$^{\text{th}}$ century, Charles Darwin postulated that all living beings share a common evolutionary history, which can be modeled via a tree [Dar59; Dar87]. Since then, the hand-drawn trees, which were based on morphological features [Dar59; Hae66; Hit40], have been replaced by trees computed using mathematical and statistical models that explain the variation among the genomic data of the species under study [Hin+15; Pen04].

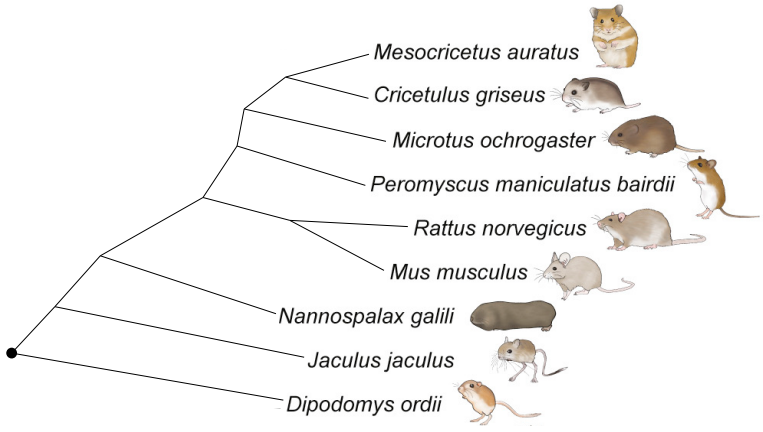


Figure 2.1 A phylogenetic tree of different species belonging to the order *Rodentia*, derived from protein-coding as well as non-coding sequences of rodent genomes [Gos+19]. ● tree root (most recent hypothetical common ancestor of *Rodentia*)

In this work, we distinguish between two types of evolutionary trees: *phylogenetic trees* describe the evolutionary history of distinct species, while *genealogical trees* describe the evolutionary relationships between individuals of the same species. For example, a tree depicting how contemporary species of the order *Rodentia* relate to each other constitutes a phylogenetic tree (Figure 2.1), while a tree depicting the evolutionary history of different *Homo sapiens* groups (subpopulations) constitutes a genealogical tree.

In phylogenetic trees, the *tips* (or leaves, i.e., out-degree 0 nodes) represent sampled extant species, whose genomic code (encoded in DNA or RNA [Alb+22, ch. 1]; Section 2.2) is known – which is predominantly the case for non-extinct species. Further, inner nodes represent hypothetical common ancestors, and edges (or *branches*), represent lines of descent. The root node (i.e., in-degree 0) of a phylogenetic tree represents the hypothetical common ancestor of all species in the tree. We call a tree with a root node *rooted*[1], and assume that all edges lead away from this root node. We call an inner node or root node of a rooted tree *bifurcating* if it has an out-degree of exactly two (Figure 2.2). If all non-leaf nodes of a tree are bifurcating, we also call the tree *bifurcating*. In contrast, we denote nodes with an out-degree exceeding two and trees containing at least one such node as *multifurcating*.[2] Multifurcating nodes can be used to model uncertainty regarding the exact ancestral evolutionary pattern in the respective part of the tree.

[1] Unrooted phylogenetic trees with undirected edges can model trees where the actual root is not known (yet) or exist because of computational reasons. However, this detail is not relevant for this work.

[2] Multifurcations are sometimes modeled using multiple bifurcations with branch lengths of zero. However, this mapping is not bijective and will influence some population genetic statistics (e.g., Lowest Common Ancestor (LCA), Section 2.4). We therefore do not use it in this work.

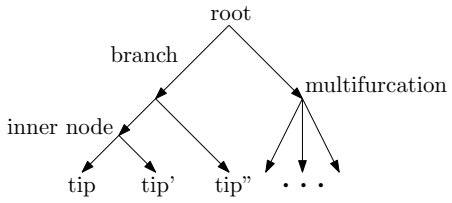


Figure 2.2 Nomenclature of an evolutionary (phylogenetic or genealogical) tree. *Tips* represent sampled genomes or fractions thereof, *branches* represent lines of descent and *inner nodes* represent hypothetical common ancestors. The *root* represents the hypothetical common ancestor of all tips. Non-leaf nodes can be *bifurcating* (out-degree of two) or *multifurcating* (out-degree exceeding two). As all edges point away from the root and towards the tips, we sometimes omit the edge arrows that indicate these edge directions.

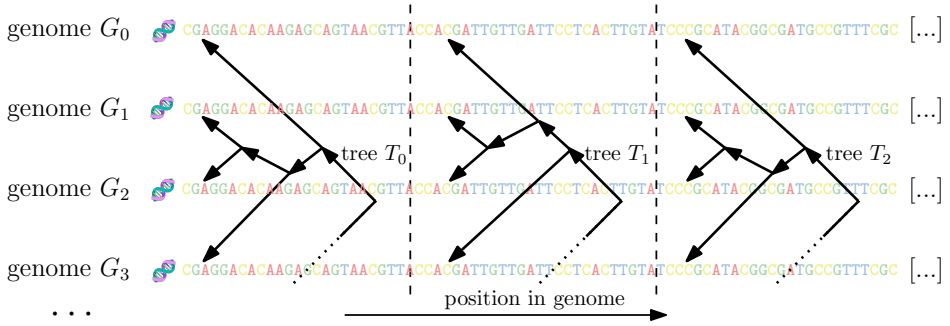


Figure 2.3 For recombining organisms such as humans, modeling the (correlated) evolutionary history of distinct, consecutive regions along the genome using a set of distinct yet correlated trees describes the evolutionary history more comprehensively than using a single tree [Hud83].

Phylogenetic research has a plethora of applications: For example, phylogenetics are applied in cancer research [Bro+17; MHS19; Rie+10; Som+17; SS17] and to study the spread [BMV01; Cas+12; CG14; Kor00; Mor+20a; Ste+20] and evolution [Bus+99] of viral infections. In forensics, phylogenetic methods are, for instance, used to trace the transmission of HIV [APV18; Ber+07; Gon14; Ou+92]. Further, phylogenetic methods are applied to predict the function of enzymes [AAZ19; ZJ12] and to aid drug design [Ash13; ATD18; BMV01]. Moreover, phylogenetic tools are employed for wild life conservation [BP94; Fer17; Hug+23; Rol+11; Vér+19], linguistics [Bow18; Gre23; Jäg18; RS18], plagiarism detection [Ryu+08], to increase crop yield [BMV01], and, of course, to infer the tree of life [Fox+80; WKW90].

In contrast to phylogenetic trees, *genealogical trees* depict the shared evolutionary history among individuals of the same species [Kel+19]. Conceptually, a genealogical tree thus further resolves the evolutionary relationships under a single tip (species) of a phylogenetic tree, down to the level of individuals. However, a single tree cannot adequately describe the processes by which individuals pass on distinct parts of their genome independently [Mor16]. We observe this, for example, in sexually reproducing species with multiple copies of the genome (e.g., two in humans[3]) [Hud83]: Here, a process called *recombination* causes the

[3] In this work, human always refers to *Homo sapiens*.

genetic material, organized in a DNA strand, to break and reconnect. Thus, the closer two genetic positions are located to each other along the genome, the higher the chance will be that they have been inherited from the same parent [SK06]. Therefore, for recombining organisms, modeling the (correlated) history of distinct regions along the genome via a *set of distinct but correlated trees* captures their evolutionary history more comprehensively than using a single tree [Hud83] (Figure 2.3). Here, each distinct tree among this set of genealogical trees, describes the evolutionary history of a distinct, consecutive region of the genome and each genomic region's evolutionary history is described by exactly one genealogical tree.

Therefore, in genealogical trees, tips represent sampled genomic material. Non-tip nodes, or *inner nodes*, represent one or multiple sequence variations inherited by its descendants. Again, non-tip nodes might be bifurcating (out-degree of two) or multifurcating (out-degree greater than two). Further, we consider all genealogical trees to be rooted, that is, they have a so-called *root* node with an in-degree of 0, and all edges leading away from it. This root represents the hypothetical set of common ancestral genetic material of all tips in the tree.

Genealogical trees are used to answer the question of how current and ancient individuals and populations of a single species, such as humans, have emerged, migrated, and how they relate to each other (*genealogy*). High-level applications of genealogical methods are deployed in medical research [All64; LH66; Oka+20; WN91], wildlife conservation [HFR20; Lui+18; SS18], and – in conjunction with recent advances in ancient DNA sequencing technology [Car+13; Mil+08; Par+15; RH09] – for studying human migration patterns over the past few thousand years [All+24; Lip+22; Par+15; Rei19].

Next to phylogenetic and genealogical trees, further models of evolution exist: For example, so-called phylogenetic networks explicitly model biological processes which a tree can not adequately describe such as horizontal gene transfers, hybridization, or gene duplication [HB05]. Moreover, consensus trees [Bry01], consensus networks [HM03], and trees-of-trees [Nye08] are different approaches aiming to capture the uncertainty associated with tree inference, summarizing over a set of plausible trees and reconciling differences.

In conclusion, evolutionary relationships between organisms can be modeled by trees. Phylogenetic trees represent the evolutionary history among distinct species, while genealogical trees describe the evolutionary relationships within a single species. Both have numerous applications, such as in disease tracking, drug design, and wildlife conservation. In this thesis, we design and implement an efficient data structure for storing genealogical trees (Chapter 3) and employ a phylogenetic tree search tool (Section 2.3) as a real-world benchmark for our software libraries ReStore (Chapter 4), ReproRed (Chapter 5), and KaMPIng (Chapter 6).

2.2 Differences in The Genetic Code Among Genomes

While the first phylogenetic trees were based on morphological features [Dar59; Hae66; Hit40], they are nowadays predominately inferred using mathematical and statistical models of evolution, which explain the genetic variation across the observed species [Hin+15; Pen04].

In this thesis, we describe the genomic code of an organism by a sequence of characters, which we call the *genomic sequence*. For instance, a sequence modeling a deoxyribonucleic acid (DNA) molecule contains four different characters: `A`(denine), `C`(ytosine), `T`(hymine), and `G`(uanine) [Alb+22, ch. 1]. Further, we describe sequences of amino acids using an alphabet of 20 characters, each representing a distinct amino acid [Alb+22, ch. 1]. Over time, mutations accumulate on these sequences, that is, characters are being replaced, inserted, or deleted [Alb+22, ch. 5]. As these mutations are inherited and thus accumulate over lines of descent [Alb+22, ch. 1], we can exploit them to compute phylogenetic and genealogical

trees (Section 2.3). A *Multiple Sequence Alignment (MSA)* is a collection of multiple genomic sequences that are arranged as rows of a matrix. One inserts gaps into these sequences such that the genetic loci (positions in the sequence) which share a common evolutionary history are located in the same column. We call the different loci of these sequence, that is, the columns of the MSA, *genomic sites*.

Genomic sites that are identical across all sampled genomes (*invariant sites*) are relevant for inferring phylogenetic trees, as they shorten the branch lengths [Lea+15]. However, invariant sites are irrelevant for computing the population genomic statistics (Section 2.4) we consider in Chapter 3, except for normalization. Hence, in this thesis, *genomic sequence* or *genome* refers to either the full genomic sequence or only genomic sites that exhibit variations across sequences, depending on the context. Further, *alleles* are concrete instances of genetic variation among genomes and encompass one or multiple (related) genomic sites. For instance, a set of genomic sequences of human individuals might have the alleles `A` and `C` at a specific genome position. We call a site with more than one allele *polyallelic*, or *biallelic* if it has exactly two alleles. Most common statistics employed in population genetics are based upon allele frequencies in a subset of the individuals in a dataset (Section 2.4).

In conclusion, nowadays, phylogenetic trees are predominantly computed on the basis of genetic variation, employing mathematical and statistical models to explain mutations. In an MSA, genomic sequences are aligned to each other such that sites with a shared evolutionary history are located in the same column. These MSAs are used, for instance, to infer phylogenetic (Section 2.3) or genealogical trees.

2.3 Phylogenetic Tree Search using the Maximum-Likelihood Method

A *likelihood* is the probability of observing given data provided a specific statistical model, including its model parameters. In the context of phylogenetic inference, this mathematical model consists of a phylogenetic tree – including its topology and branch lengths – as well as a statistical description of the evolutionary forces acting on a genomic sequence [Fel78]. We denote the latter as *evolutionary model*. It describes the prior frequencies of the genomic states at the hypothetical ancestor, how probable different mutations are, and also accounts for the phenomena that different parts of the genetic sequence mutate at different rates. The observed data in our likelihood evaluation are the aligned sequences in the MSA (Section 2.2).

Given a phylogenetic tree and an evolutionary model, we can compute the *likelihood score* for each site in the MSA. For numerical reasons, we compute the logarithm of this score, which we call the *Log-Likelihood (LLH)-score*. Summing over the LLH-score of each MSA site yields a single LLH-score for a given tree and evolutionary model. While optimizing the tree topology is $\mathcal{NP}$-hard [CT05], numerically optimizing the branch lengths and evolutionary model is comparatively fast [Sta04]. Thus, we often refer to the LLH-score of a phylogenetic tree, assuming that the remaining model parameters have been optimally chosen.[4]

Maximum-likelihood based phylogenetic inference aims to infer *the* most likely tree among all possible trees [Fel81]. However, Chor and Tuller show this to be an $\mathcal{NP}$-hard problem [CT05]. Therefore, algorithms that shall analyze real-world datasets have to employ heuristics [Koz+19; Ngu+15]. Nonetheless, likelihood-based phylogenetic inference has been repeatedly shown to be able to recover the true tree on simulated sequence data [GG03; KF94; WM03]. For example, the maximum-likelihood based phylogenetic inference program RAxML-NG [Koz+19] employs a greedy hill-climbing algorithm to search the tree topology

[4] Note that branch length optimization can also converge to distinct local maxima.

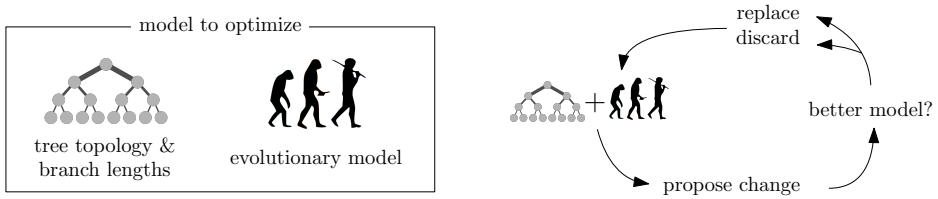


Figure 2.4 Simplification of a heuristic maximum-likelihood based tree inference algorithm. The algorithm iteratively improves upon the currently best known model by proposing changes to the tree topology, followed by a numerical optimization of the branch lengths and evolutionary model parameters. If the LLH-score of the new model is higher, it replaces the former best known model as starting point for further optimization.

space. That is, the currently best-known tree is iteratively improved by proposing topologically similar trees, and numerically optimizing the branch lengths and evolutionary model. It then keeps the new topology if it has a higher LLH-score (Figure 2.4).

In summary, likelihood-based phylogenetic tree inference heuristically searches for the best phylogenetic tree it can find. We sometimes call this the *best known Maximum-Likelihood tree*. As optimizing the tree topology is $\mathcal{NP}$-hard [CT05], we employ greedy algorithms to iteratively refine the tree topology by proposing minor changes and adopting them if they improve the tree's overall likelihood score.

2.4 Statistics in Population Genetics

In contrast to phylogenetic trees (Section 2.3), the genealogical trees we consider in Chapter 3 are either inferred from empirical data using a hidden Markov-chain [Kel+19], or simulated using methods outside the scope of this thesis [Kel+18]. In Chapter 3, we describe a novel data structure to store sets of genealogical trees which substantially accelerates the computation of statistics that are commonly used in the field of population genetics (Chapter 3).

When computing statistics on genealogical trees and the sequences associated with these, we often only select a subset of the samples (tips in the genealogical tree) for a given query. We call such a selection the *selected* samples, or *sample set*. The exponentially large number of possible sample sets yields pre-computing all statistics of interest infeasible, and therefore necessitates algorithmic improvements.

Many of the statistics that are commonly used in population genetics are functions of the *allele frequencies* [RTK20], that is, how often each allele (Section 2.2) occurs in the sequences of a sample set. For example, the sequence *diversity* [NL79] reflects the probability that two random sequences differ at a random site (both chosen uniformly at random). *Segregating sites* exhibit more than one allele (i.e., polyallelic sites, Section 2.2). The *Allele Frequency Spectrum (AFS)* [Fis31] is a histogram over the allele frequencies. Some statistics also operate on multiple sample sets. For example the sequence *divergence* [NL79] is the probability that two sequences – each chosen uniformly at random from a distinct sample set – differ at any given site. We can derive more elaborate statistics from the above basic statistics, using up to four disjoint sample sets. Examples include *Patterson's* $f_{\{2,3,4\}}$ [Pat+12], *Tajima's D* [Taj89], and the *Fixation Index* F_{ST} [Wri50]. While we define these statistics on a per-site basis, we often average them across many or all sites in the input genome or within a sliding window over all sites of the genome.

Tajima's D statistic can, for instance, be used to detect genes (parts of the genome encoding a RNA strand or protein) which did not evolve neutrally. One interpretation of

non-neutral evolution is that the gene in question encodes a protein which is advantageous for the fitness of the respective organism [EWV18]. Tajima's D has also been used to study the interaction of pathogens and the organisms they infect [EWV18; Mon+13]. Additionally, Tajima's D can detect genetic bottlenecks, that is, that a sampled population has been substantially smaller in the past. This can be caused, for example, by an environmental change or by a small set of individuals spreading to a new habitat [GJB13]. The Fixation Index F_{ST} is used, for example, to estimate how many individuals migrate between populations, identify regions that are under selection, or to provide a null-distribution when matching a genetic profile in a forensic analysis [HW09]. Patterson's f statistics are used, for example, to test whether a population descended from one or multiple ancestral populations, and to determine their proportions [Pet16; Rei+09]. Further, Patterson's f is employed to find the closest relative to a given population [Pet16; Rag+13] and to model complex non-tree like population structures for sexually reproducing organisms [Pet16; Pet21].

The Lowest Common Ancestor (LCA) [AHU73] of two or more tips in a genealogical tree is the lowest (farthest from the root) inner node on *all* paths from the root to the tips. In population genetics, the LCA is sometimes also called the Most Recent Common Ancestor (MRCA). Population geneticist employ LCA queries on genealogical trees, for example, to answer questions such as "Did the evolutionary history of genomes of group A and B separate at the time this land bridge vanished?" This involves computing the LCA of all genomes in A and B and estimating the time of the resulting ancestor using methods beyond the scope of this thesis [AM20].

In summary, statistics derived from differences between the genomic sequences of sampled organisms are widely used in population genetics. For example, they can help to identify genetic bottlenecks, migration between populations, and can be used to uncover close relatives of a given population. Further, the LCA, which depends on the topology of the genealogical tree is, for instance, used to date population splitting events.

2.5 Parallel Machine Model

Modern High Performance Computing (HPC) environments have single- or multi-CPU machines (or *compute nodes*), with each CPU having multiple cores that all participate at the computation (Figure 2.5). CPU cores on the same physical machine communicate with each other via shared memory, while CPU cores on different machines communicate over an interconnection network, for example, InfiniBand [Las+09], OmniPath [Bir+15], or TCP/IP [IETF22; NWG98]. Further, we often employ the common term Processing Element (PE) [San+19, ch. 2.4] to abstract over specific hardware setups. Here, one PE maps to a single CPU core in real hardware.

In this thesis, PEs communicate via messages that are exchanged either via shared memory – if they reside on the same machine – or over the interconnection network. For this communication, we employ the MPI interface [MPI4.1], which also offers collective communication operations in addition to point-to-point messages, as well as synchronous and asynchronous messages (Section 2.6).

When reasoning about the running time of a parallel algorithm (e.g., in Chapters 4 and 5), we consider the time spent for communication in addition to the time spent on performing computations. In this thesis, we assume that messages have exactly one sending and one receiving PE and that exchanging a message of length l requires time $\alpha+\beta\cdot l$ [San+19, ch. 2.4]. Here, α represents the *startup overhead* (latency) for establishing a connection associated with this message. Further, β represents the time to transfer a single byte. On common

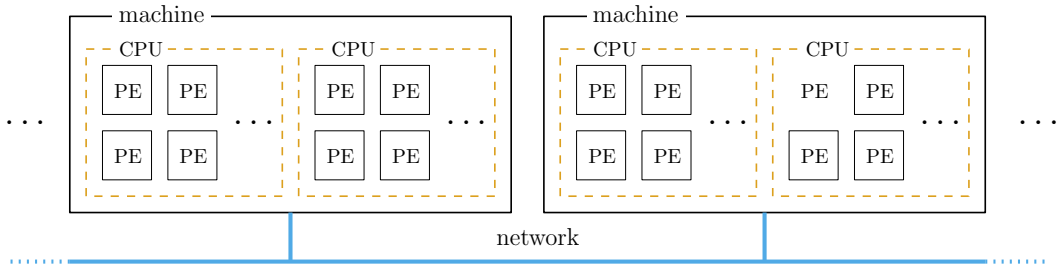


Figure 2.5 A distributed-memory machine as typically used in High Performance Computing. Each machine, or *compute node*, consists of one or multiple CPUs, each with multiple cores, which we often refer to as abstract Processing Elements (PEs). PEs on the same physical machine communicate via shared memory, while PEs on distinct machines communicate via a network.

distributed-memory machines, $\beta \ll \alpha$ holds. Therefore, when evaluating the running time of parallel algorithms, we consider two important factors: The bottleneck *number of messages* that are being sent and received, and the bottleneck *communication volume*. The bottleneck number of messages that are sent and received during a run describes the maximum number of messages sent or received by a single PE. This influences parallel performance because of the startup overhead associated with each message. As a result, sending or receiving data from one PE to another via a single message is typically faster than splitting that data into many parts and sending each part to a different receiver. The bottleneck communication volume describes the maximum amount of data sent or received by a single PE and represents a point on the critical path of the application.

In summary, modern HPC systems consist of compute nodes with single or multi-CPU architectures, where each CPU has multiple cores, which we abstract as PE. In this work, PEs communicate via messages they pass over the shared memory inside a machine or over a fast interconnection network between machines. When analyzing the runtime of our algorithms, we consider the communication overhead next to the amount of work. Here, we distinguish between the startup overhead associated with each message and the time for transferring the actual data.

2.6 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) [MPI4.1] is *the* message passing standard used in distributed scientific computing. In addition to point-to-point messages, MPI includes [MPI4.1] directives for synchronization and collective operations [San+19, ch. 13]. For example `MPI_Allreduce` takes as input one value per PE as well as a reduction operator [MPI4.1, ch. 6.9.1]. RAxML-NG then returns the result of the reduction over all input values at all PEs. Major contemporary implementations of MPI include OpenMPI [Gra+06], MPICH [MPICH], and Intel MPI [Int25].

Moreover, MPI offers synchronous and asynchronous communication directives [MPI4.1, ch. 3.4]. A synchronous send operation does not return until the corresponding receive operation on the remote PE has been completed, and the respective data have been sent. In contrast, an asynchronous send operation does not wait for the sender to complete the data-transfer, allowing the message exchange to occur in the background and thereby overlaps computation with communication. We employ asynchronous communication, for example, in the implementation of our reproducible reduction algorithm (Chapter 5) in order to receive messages in the background while performing the reduction on locally available data.

In conclusion, MPI offers point-to-point messages as well as abstract collectives, such as all-to-all. Next to synchronous send operations, which block until the respective receive operation has been completed, MPI also offers asynchronous operations, which allow programs to overlap communication with computation.

2.7 Hardware Failures and Failure-Tolerant Algorithms

Machines in HPC systems experience failures, for example, due to core hangs, kernel panics, file system errors, file server failures, memory corruption, network outages, air conditioning failures, or power halts [Gup+17; Lu13]. Current petascale systems fail about 0.8 to 5.7 times per day: For example Martino *et al.* [Mar+14; Mar+15] report 2.8 to 5.7 failures/day on the Blue Waters Cray XK7 supercomputer that comprises 22 640 compute nodes. Further, Gamell *et al.* [Gam+14] found that the ORNL's Jaguar Titan Cray XK7 system experiences on average 2.3 failures/day. Moreover, Gupta *et al.* [Gup+17] report on four petascale systems (Cray and Intel) containing up to 18 688 nodes (560 640 cores), each failing on average 1 to 3.4 times/day. More recently, Rojas *et al.* [Roj+21] report 3.1 failures/day over a period of five years on a Cray XK7 machine comprising 18 688 nodes [Roj+21] while Di *et al.* [Di+19] observe an average failure rate of 1.3 days on a 49 151 node (786 432 core) IBM Blue Gene/Q PowerPC system. Further investigations into the failure rates of HPC systems were performed by Hacker *et al.* [HRC09], El-Sayed and Schroeder [ES13], Schroeder [SG10], Sahoo *et al.* [Sah+04], Oliner and Stearley [OS07], Bhagwan *et al.* [BSV03], Chen and Sun [CS17], and Liang *et al.* [Lia+06]. In upcoming systems, we expect a hardware failure to occur every 30 to 60 minutes [Cap+14; DHR15; Sni+14]. Thus, handling such failures constitutes a major challenge for future exascale systems [SDM10].

In this thesis, we consider *fail-stop* failures, that is, the respective hardware and processes running on them either communicate their failure state or simply stop responding. We consider these processes as *failed* for the remainder of the program execution. We call the remaining nodes, which can continue with the computation, *surviving* nodes. Other types of failures not considered in this thesis include transient failures and silent data corruption [DHR15]

Various scientific applications have been made failure-tolerant, for example, a linear and partial equation solver [Ali+16], a plasma simulation [Obe+17], a molecular dynamics simulation [EG03], and a Fast Fourier Transformation [Koh+17]. In previous work, we increased the checkpointing frequency of RAxML-NG, a widely used phylogenetic inference tool (Section 2.3), in order to yield it more resilient with respect to hardware failures [Hüb+21b].

Four of the main techniques for yielding programs failure-tolerant are: Algorithm-Based Failure-Tolerance (ABFT), restarting failed sub-jobs, redundant computations, and checkpoint/restart. ABFT requires the algorithm in question to be extensible such as to include redundancy, for example, by adding additional rows to a matrix when solving a linear equation, and is thus used predominantly in numerical applications [Bos+08; VM97]. Restarting failed sub-jobs is feasible as a failure mitigation strategy when the program at hand can be split up into separate small, short, and well-defined work packages that can easily be redistributed among nodes and managed via a (possibly distributed) work queue. For instance, some MapReduce frameworks implement this approach [Mem+16]. Another failure-tolerance strategy is to schedule each computation multiple times on different hardware [HZ15]. While this is easy to implement and induces nearly no runtime overhead, it requires, however, substantial additional computational resources. Finally, failure-tolerance using the *checkpoint/restart* strategy involves to frequently write the program's state to checkpoint files or

system memory. Thus, after a failure, the program can subsequently restore this previous state from the most recent checkpoint and restart the computation from there.

Following a failure, an application has to redistribute the work previously performed by a failed PE using either the shrink or the substitute strategy [AHE18]. Under the *substitute* strategy, a replacement PE takes over the work previously performed by the failed PE. This circumvents the need for re-balancing the workload and simplifies (re-)loading the required data. However, reserving idle processors for this purpose constitutes a waste of resources. In contrast, when employing the *shrink* strategy, the program's load balancer (re)distributes the work performed by the failed PE among the remaining, or *surviving*, PEs. The shrink strategy does therefore not require spare PEs but reloads fractions of the data on many or even all PEs. While the number of failures an algorithm can tolerate using the substitute strategy is limited to the number of spare PEs; this limitation does not apply to the shrink strategy [AHE18]. While many checkpointing libraries target the substitution strategy [Aga+04; Bar+17; Bau+11; BH14; Gam+14; Lu05; Moo+10; Nic+19; Sha+19; TH14], we argue for using the shrinking strategy as it does not waste computational resources and avoids the communication bottleneck of one PE receiving all data previously held by a single failed PE (Chapter 4).

In summary, petascale HPC systems experience about 0.8 to 5.7 failure/day for a plethora of reasons, including hardware and software causes, of which we consider those inducing fail-stop failures. As upcoming HPC systems are predicted to fail approximately once an hour, various scientific applications have been made failure-tolerant using strategies such as ABFT, re-scheduling failed sub-jobs, or checkpoint/restart. During failure-recovery, the program can either acquire replacement for the failed PEs or shrink to the surviving PEs, avoiding wasted computation resources but necessitating fine-grained data recovery mechanisms.

2.8 Checkpoint/Restart

Checkpoint/restart approaches are the de-facto standard failure-mitigation strategy in scientific codes [DHR15]. They can be classified into *system-level* and *application-level* approaches. System-level approaches, for example the Berkeley Lab Checkpoint/Restart (BLCR) library, work at the level of the operating system and create a backup copy of the full memory footprint, register content, call stack, and other properties of the application process [DHR15]. Thus, system-level approaches are (almost) transparent to the application, but are also agnostic to which subset of the program's memory is crucial and which subset can be recomputed at low computational cost [HD06; Rom02]. For instance, in any tree inference program that conducts phylogenetic likelihood calculations (Section 2.3), the intermediate results of these likelihood computations dominate the memory requirement of the program and can require terabytes of memory in large-scale analyses [Jar+14; Mis+14]. However, as these intermediate results can be efficiently recomputed, they should not be saved in a checkpoint [Hüb+21b]. Conversely, application-level checkpoints allow the application to specify which sections of its memory shall be backed-up, reducing the amount of data that has to be backed-up drastically [Hüb+21b].

Checkpoint/restart strategies can also be divided into *coordinated* and *uncoordinated* checkpointing approaches [DHR15]. In coordinated checkpointing strategies, all PEs write their checkpoints at the same time. While this comes at the cost of additional synchronization, it also simplifies implementation. In uncoordinated checkpointing, PEs do not synchronize when creating checkpoints, implying increased software complexity for ensuring that consistent checkpoints are created. As synchronization is cheap in modern HPC systems with

high-bandwidth, low-latency interconnections, Gavaskar and Subbarao [GS13] recommend coordinated checkpointing when developing applications for these systems.

Further, checkpointing libraries usually write their checkpoints to a parallel file system (PFS) [Aga+04; Bau+11; Nic+19; Sha+19], implying slow recovery due to low disk access speeds and high congestion as many processors simultaneously access the same resources. In contrast, *in-memory* checkpointing libraries keep all of their checkpointing data in the main memory of the PEs. To the best of our knowledge, there exists no general purpose checkpointing solution allowing for in-memory recovery without requiring spare cores.

In summary, the prevalent [DHR15] failure-mitigation strategy in scientific computing on HPC system is checkpoint/restart, which can be further classified into application-level/system-level libraries, if checkpoints are coordinated/uncoordinated among the PEs, and if checkpoints are created in-memory or written to the PFS.

2.9 Failure-Tolerance in the MPI Standard

The upcoming MPI standard 5.0 will support mechanisms to mitigate failures of compute nodes and/or network components. Currently, there exist two actively developed MPI implementations which already support MPI's failure-mitigation mechanisms: MPICH [Gro02] as of version 3.1 and User Level Failure Mitigation (ULFM) [Bla+13], which has now been merged into OpenMPI 5. Various failure-tolerant scientific software have already been developed using these MPI implementations [Ali+16; EG03; Koh+17; Lag+16; Obe+17].

The exact mechanism used to detect a hardware failure depends on the type of the interconnection network. One example are heartbeat signals: PEs send messages, so-called heartbeat signals, at regular intervals to each other in order to indicate that they are still alive [Hüb+21b]. If a PE has not sent any heartbeat signal for a specified amount of time – the heartbeat timeout – its observer will report its failure. The failure is subsequently reported to the application when it tries to communicate with the failed PE [ULFM]. The application can subsequently handle the failure, for example, by asking MPI to create a new communicator without the failed PE(s), recovering lost application data, re-distributing the work formerly assigned to the failed PE(s), and finally rolling-back its state to the last checkpoint, restarting from there.

We design and implement our checkpointing library ReStore (Chapter 4) for these MPI failure-mitigation mechanisms. Additionally, we verify the correctness of our implementation using ULFM as failure-tolerant MPI implementation and introduce an abstraction over this failure-mitigation interface to our C++ MPI wrapper KaMPIng (Chapter 6).

2.10 Computational Reproducibility

Next to disseminating results, scientific publications also serve the purpose to convince the reader of the correctness of this result [Mes10]. However, even though reproducibility is considered to be an essential requirement for validating scientific claims [IT18], efforts to reproduce findings in psychology [Aar+15], cancer research [BE12], or pharmaceutics [PSA11] succeeded in only 11 to 39 % of studies.

In computer science, common reproducibility practices include documenting and archiving the source code and input data. Further, either automation or detailed instructions on how to perform the experiments are necessary. This documentation should, for instance, include the software environment (e.g., compiler settings, operating system, and random seeds) as

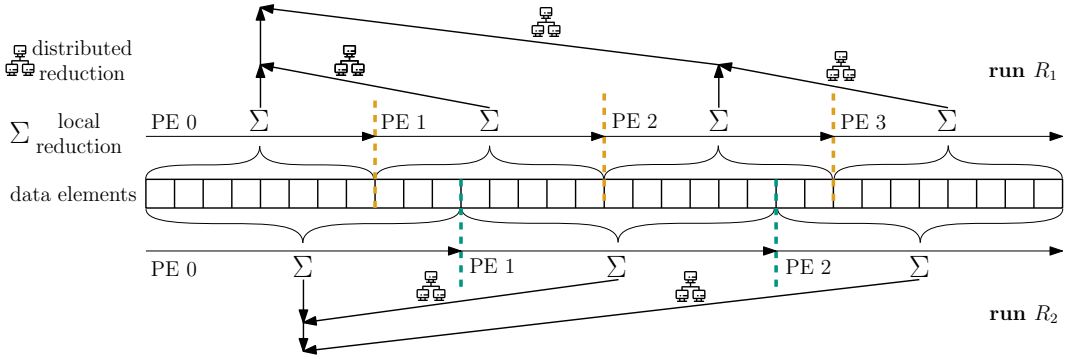


Figure 2.6 Common implementations of distributed-memory parallel reduction algorithms are not reproducible if the number of PEs changes between runs. First, each PE reduces its local elements, arriving at a single intermediate result per PE. Next, these intermediate results are exchanged via messages over the network and further reduced into a single element. Note that the number of PEs influences the order of reduction operations and thus how results are being rounded.

well as the hardware being used [CK18; IT18]. In practice, however, Collberg *et al.* [CPW15] were unable to even obtain and build the source code of nearly half of 402 ACM papers.

Even if we are able to obtain and build the source code and data, the program might arrive at slightly different results for floating-point calculations on different CPU architectures [GL99], or when using a different number of PEs [She+20a]. For instance, in phylogenetics, Darriba *et al.* [DFS18] and Shen *et al.* [She+20a] report, that the CPU type and number of PEs used to perform a phylogenetic inference can influence the final result, even if all other parameters (e.g., the random seed) are identical. In this work, we are concerned with this *bit-reproducibility*, which we define as a program producing bit-wise identical results when run twice on the same input data and under the same settings.

Next to the software and hardware environment, some codes experience timing effects as an additional source of irreproducible behavior [GH22; SS24]. Here, the point of time when a given message arrives at a specific PE influences the exact search path a heuristic takes. However, many codes, for example RAxML-NG, which we consider in Chapter 5, do not exhibit this behavior.

Tools such as containerization exists to archive the software environment and settings used for experiments [Boe15]. In contrast, hardware is substantially more difficult to archive [IT18]. However, scientific codes which yield bit-reproducible results in a multitude of hardware environments, including different CPU types and varying core-count, allow us to reproduce scientific findings without access to the exact hardware originally used.

The root cause for non-reproducible floating-point results is that fundamental arithmetic operations such as additions or multiplications on IEEE 754 [IEEE754] floating-point numbers induce rounding errors [Gol91]. On petascale supercomputers, this implies up to 10^{18} rounding operations per second. One factor determining the rounding of intermediate results are the specific CPU instructions which the compiler decides to use. For example, Fused Multiply-and-Add (FMA) instructions perform one multiplication and one addition, rounding the result only once, instead of twice: Once after the multiplication and once more after the addition [Int24, ch. 14.5.2]. Further, as different Single Instruction Multiple Data Stream (SIMD) variants (e.g., AVX, SSE3, etc.) offer different instructions and vector widths, respective math-kernels often yield non-identical results, with examples including the Intel math kernel library oneMKL [Int25a], the math functions included in the Intel compiler [Int25b], and likelihood

computation in phylogenetics [She+20a]. In this thesis, we assume that the same instructions are carried out in each program execution. This can be achieved by, for example, archiving the compiled application binary or by passing the respective reproducibility flags to the compiler in addition to avoiding run-time discernment of detected hardware [CK18].

In addition to executing identical CPU instructions, we also need to ensure that floating-point related CPU settings are consistent between runs: For example, the IEEE 754 rounding mode [Int24, ch. 4.8.4] could be set to always round to the nearest even number at each program launch. Further, denormalization of floating-point numbers – a special representation for values close to zero [Int24, ch. 4.8.3.2] – has to be consistently enabled or disabled [Int24, ch. 10.2.3.4]. Additionally, as speculative execution might inconsistently throw floating-point exceptions that occur in miss-predicted paths because of timing effects, the floating-point exceptions have to be masked [CK18]. If we use the 80 bit wide x87 registers[5], we have to ensure that their precision setting [Int24, ch. 8] is set to using 64 bit precision, thereby keeping it consistent and avoiding truncation when flushed on a context switch.

By using the same application binary (identical CPU instructions) and ensuring that the floating-point settings of the CPU are consistent between runs, we can therefore (nearly) ensure bit-identical results. However, in common distributed-memory parallel implementations of the reduction algorithm, the number of involved PEs will also influence the operation order and thus, the end-result [Li+23; SMR24] (Figure 2.6). This is due to the fact that in parallel algorithms, a large number of values is often distributed across all PEs, with each contributing a part of the data to the reduction. Changing the parallelization level often induces a different data distribution across the PEs as well as a different communication pattern, thereby changing the order in which operations are computed.

Reports of these low-level discrepancies in floating-point values affecting high-level results of scientific codes exists in the fields of phylogenetics [DFS18], sheet metal forming [Die12], fluid simulations [Wie+19], climate and weather modeling [DN15; HD01], power grid analysis [Vil+09], atomic and molecular dynamics [Cle+13; Tau+10], as well as fluid dynamics [RRA11]. In these examples, the high-level differences were caused by iterative algorithms that converge to different local optima. In Chapter 5, we analyze the reproducibility of the phylogenetic tree inference (Section 2.3) tool RAxML-NG on ten thousand datasets, develop a bit-reproducible distributed-memory reduction algorithm and integrate it into the C++ MPI wrapper KaMPIng (Chapter 6). We also create a bit-reproducible fork of RAxML-NG.

In conclusion, reproducibility of scientific results constitutes an important pillar of scientific publications. However, efforts to reproduce results often exhibit low success rates. In computer science, computational reproducibility can be substantially improved by archiving the software environment, application settings, compiled binary, and datasets used. However, the CPU type and core-count induces floating-point rounding errors, which can cause discrepancies in high-level results. Therefore, reproducibility requires consistency of CPU floating-point settings across runs as well as bit-reproducible reduction algorithms for parallel codes.

[5] As the Linux [Lu+24, ch. 3.2.3] as well as the Microsoft [Mic22] x64 calling conventions both do not employ the x87 registers to pass parameters anymore, they are no longer commonly used by compilers.

3 Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests

Attribution: This chapter is based on the peer-reviewed publication "Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests" [HS24b], by Alexandros Stamatakis and myself. Large parts of this chapter have been copied either in modified or verbatim form from this publication, where I am the first author.

Summary: The field of population genetics attempts to advance our understanding of evolutionary processes. It has applications, for example, in medical research, wildlife conservation, and – in conjunction with recent advances in ancient DNA sequencing technology – studying human migration patterns over the past few thousand years. The basic toolbox of population genetics includes genealogical trees, which describe the shared evolutionary history among individuals of the same species. They are calculated on the basis of genetic variations. However, in recombining organisms, a single tree is insufficient to describe the evolutionary history along the entire genome. Instead, a collection of correlated trees can be used, where each describes the evolutionary history of a distinct consecutive region of the genome. The current state of-the-art data structure for modeling this, tree sequences, compresses these genealogical trees via edit operations when moving from one tree to the next along the genome – instead of storing the full, often redundant, description for each tree. We propose a new data structure, genealogical forests, which compresses the set of genealogical trees into a Directed Acyclic Graph (DAG). In this DAG, identical subtrees that are shared across the input trees are encoded only once, thereby allowing for a straight-forward memoization of intermediate results during the computation of population genetic statistics. Additionally, we provide a C++ implementation of our proposed data structure, called `gfkit`*, which is* 2.1 *to* 11.2 *(median* 4.0*) times faster than the state-of-the-art tool (*`tskit`*) on empirical and simulated datasets when computing important population genetic statistics such as the Allele Frequency Spectrum, Patterson's f, the Fixation Index, Tajima's D, pairwise Lowest Common Ancestors, and others. On Lowest Common Ancestor queries with more than two samples as input,* `gfkit` *scales asymptotically better than the state-of-the-art, and is therefore up to three orders of magnitude faster. In conclusion, our proposed data structure compresses genealogical trees by storing shared subtrees only once, thereby enabling straight-forward memoization of intermediate results, yielding a substantial runtime reduction and a potentially more intuitive data representation with respect to the state-of-the-art. Our improvements will boost the development of novel analyses and models in the field of population genetics and increase scalability to ever-growing genomic datasets.*

3.1 Introduction

The evolutionary history of living beings can be modeled using a tree [Dar59; Dar87], which is nowadays commonly computed using mathematical and statistical models that account for variations among the genomic data of the observed species [Hin+15; Pen04] (Section 2.1). Next to analyzing the evolutionary history among distinct species (*phylogenetics*), the question of how living and past individuals and populations of a single species, for example humans,[1] have emerged, migrated, and how they relate to each other (*genealogy*) is also of interest. Population genetics is the sub-field of evolutionary biology that addresses these questions. It advances our understanding of evolutionary processes such as mutations, genetic drift, gene flow, and natural selection [Lui+18]. Its downstream applications include for example medical research [All64; LH66; Oka+20; WN91], wildlife conservation [HFR20; Lui+18; SS18], and – in conjunction with recent advances in ancient DNA sequencing technology [Car+13; Mil+08; Par+15; RH09] – the study of human migration patterns over the past few thousand years [All+24; Lip+22; Par+15; Rei19] (Section 2.1).

3.1.1 Genealogical Trees, Tree Sequences, and Forests

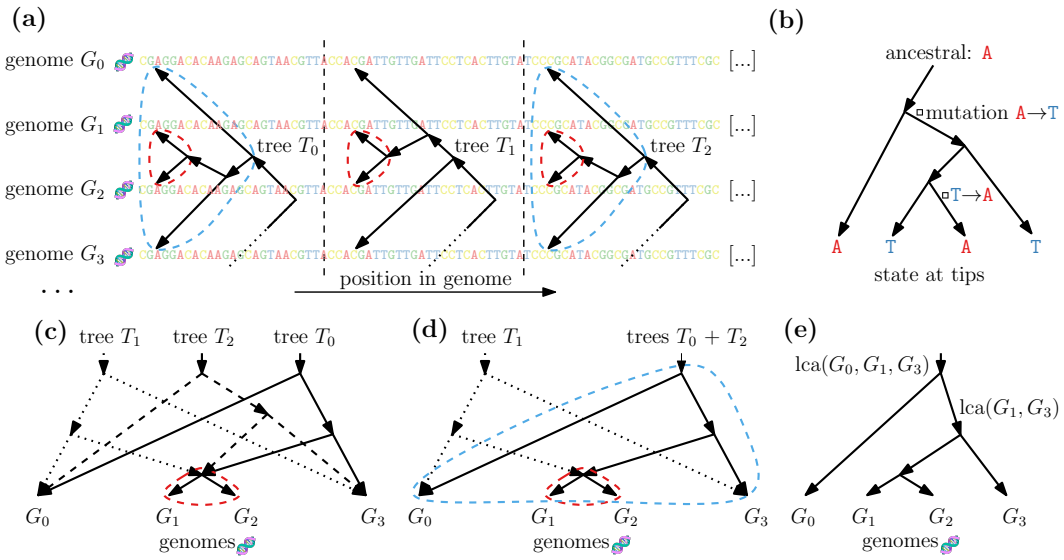


Figure 3.1 **(a)** Different, but correlated, trees each describe distinct regions of aligned genomes from distinct individuals. The subtree (G_1, G_2) (red) appears in all displayed trees, whereas the subtree $(G_0, ((G_1, G_2), G_3))$ (blue) appears only in T_0 and T_2, but not in T_1. **(b)** A tree can be annotated to describe the genetic states of the genomes at its tips by storing ancestral states and corresponding mutations annotated at the tree edges (we show only a single genomic site). **(c)** `tskit` encoding of the three trees from (a). `tskit` re-uses (G_1, G_2) but not $(G_0, ((G_1, G_2), G_3))$, as the latter does not re-occur in *consecutive* trees. **(d)** Our proposed encoding (`gfkit`) of the trees from (a). `gfkit` re-uses both, (G_0, G_1) when describing T_1, and $(G_0, ((G_1, G_2), G_3))$ when describing T_2. **(e)** Lowest Common Ancestor of two (lca(G_1, G_3)) or more (lca(G_0, G_1, G_3)) selected samples.

Genealogical trees depict the shared evolutionary history among genomes of distinct individuals from the same species and are based on the genetic variation across sampled

[1] In this work, human always refers to *Homo sapiens*.

individuals [Kel+19] (Section 2.1). As single trees cannot adequately describe the evolutionary history of recombining organisms [Hud83; Mor16], we model the (correlated) history of distinct regions along the genome using distinct (correlated) trees [Hud83] (Section 2.1).

Kelleher *et al.* [Kel+19; KEM16; RTK20] describe *tree sequences* (Figure 3.1.a; implemented in `tskit`), which are collections of correlated genealogical trees, each describing the evolutionary history of a window of adjacent sites along the genome. In the genealogical trees that are stored in tree sequences, tips represent biological samples, edges represent lines of descent, and mutations are meta-data associated with these edges (Figure 3.1.b-c). In tree sequences, the edges are valid only for a specified region of the genome, and describe the evolutionary history of this region. Statistics over the topologies of genealogical trees and differences between the genomic code associated with the tips of the genealogical trees are frequently deployed in population genetic studies to conduct quantitative assessments, for instance, how genetically diverse a population is (Section 2.4).

3.1.2 State of the Art and Contribution

Tree sequences model the shared evolutionary history of the genomic samples associated with their tips. They allow for reducing the storage space required for storing the set of genomes associated with their samples as well as to reuse some computations in statistical queries [Kel+19; KEM16; RTK20]. For instance, the human chromosome 20 in the Thousand Genomes Project (TGP) collection (Section 3.5) contains 5008 sequences with 860 thousand variant sites; each sequence being in one of four potential states (`A`, `C`, `T`, or `G`) at each site [Kel+19]. This results in a theoretical space requirement of 1 GiB if stored on a base-by-base basis using 2 bit per sequence and site – compared to a corresponding tree sequence file size of 283 MiB [Kel+19].

Tree sequences store the evolutionary history of each part of the genome via the corresponding genealogical trees and allow reusing partial results between correlated, yet topologically distinct trees. They thus avoid some – yet not all possible – redundant computations [KEM16; RTK20]: Tree sequences can reuse only intermediate results for shared subtrees among *adjacent* trees along the genome [KEM16; RTK20]. They do not allow reusing intermediate results across trees that are further apart, if the respective subtree is not part of all intermediate trees (Figure 3.1.c). Tree sequences use edit operations (edge insertions and removals) in order to describe changes/differences in the tree topology when moving from one tree to the next along the genome; and thus across recombination events. Therefore, determining if a given subtree was already present in an earlier tree, is not trivial. For example, if an edge from node a to node b ($a \to b$) is removed and a (sans meta-data) identical edge $a \to b$ is added back a few trees further down the genome, other edges might have changed in the subtree below node b, thereby invalidating the intermediate results for b.

In contrast, the core concept of our novel data structure is to encode and store shared subtree across *all* (not just adjacent) trees exactly once. This enables queries on the trees' topologies (Section 3.3.3) and genomic sequences (Section 3.3.4) to reuse intermediate results across *all* trees (*memoization*).

3.1.3 Outline of this Chapter

After providing an overview of related work (Section 3.2), we describe the core idea of our proposed data structure `gfkit` (Section 3.3), as well as the implications of our design decisions. Further, we describe `gfkit`'s query algorithm for computing the Lowest Common Ancestor (Section 3.3.3) and other common statistics used in population genetics (Section 3.3.4). We

also describe the conversion of the `tskit` to the `gfkit` data structure (Section 3.3.6) as well as two variants of our data structure (Section 3.3.5 and Section 3.3.7). Further, we evaluate the performance of `gfkit` (Section 3.6): We report speedups (Section 3.6.1 and Section 3.6.2) and analyze the created Directed Acyclic Graphs (Section 3.6.3 and Section 3.6.4). Next, we discuss the space usage of `gfkit` vs. `tskit` (Section 3.6.6) and describe our qualitative observations concerning the numerical stability of the computed statistics (Section 3.7). We conclude, and highlight possible directions of future work in Section 3.8.

3.2 Related Work

The genomic code of related species or individuals of the same species is highly redundant, even when only considering variant sites [RTK20]. Ané and Sanderson [AS05] thus propose to compress related genomic sequences using a phylogenetic tree. Here, the tips of the tree represent the genomic sequences, which are fully described by storing the ancestral state for each site at the tree root and the respective mutations along the edges of the tree (Figure 3.1.b). Ané and Sanderson primarily use a biologically reasonable evolutionary tree with an optimized parsimony score in order to attain a good compression ratio; they do not consider the tree to explicitly model the evolutionary history.

Kelleher *et al.* [Kel+19; KEM16; RTK20] introduce *tree sequences*, which model the evolutionary history of a set of related genomes, allow reusing some intermediate results when computing statistics on them, and enable space-efficient storage of these sequences (Section 3.1.2) by also storing them implicitly using ancestral states and mutations.

Matthews and Williams [MSW10] compress a collection of trees by encoding each *bi-partition* exactly once. Here, a bipartition is a split of a tree's tips into two disjoint sets. Note that each edge of a tree induces a bipartition and thus the set of all bipartitions fully describes a tree. However, this does not allow for direct access to the tree topologies, which we require to compute the Lowest Common Ancestor (LCA, Sections 2.4 and 3.3.3).

Directed Acyclic Graphs (DAGs) only contain directed edges and no path of length > 0 from a node to itself. The tree terminology introduced in Section 3.1.1 generalizes to DAGs: We call out-degree 0 nodes *tips*, which represent samples and their associated genomes. In contrast to trees, DAGs may have multiple *root* nodes, that is, nodes with an in-degree of 0.

Theoretical computer science has studied the compression of single trees via DAG-compression [Bil+11; Sak09], tree grammars [DST80], and top-trees [Bil+15]. DAG-based compression reuses identical (topology and label) subtrees, when they occur multiple times in the same tree. In a tree, each node also induces/represents the subtree containing it and all of its descendants. Thus, instead of encoding a subtree a second time, we add an edge to the node representing the already encoded subtree, resulting in a DAG. In a single genealogical tree, all tips (representing samples) are distinct and thus not compressible via DAG-based compression. However, one can extend the idea of representing each unique subtree only once to sets of trees (*forests*), by reusing subtrees that form part of multiple trees (Section 3.3). Ingels [IA20; Ing22] implements this idea by encoding a forest as a DAG. However, they focus on the enumeration of trees instead of on reusing intermediate results during computations. Tree grammars and top-tree based compression reuse tree patterns, that is, any identically-connected subgraph (not necessarily including all descendants) [Bil+15]. It remains an open question if top-trees[2] can encode the set of related genealogical trees while supporting the required queries efficiently.

[2] Minimal tree grammars are $\mathcal{NP}$-hard to construct and do not support efficient navigation [Bil+15].

Trefzer and Stamatakis [TS18] use the balanced parenthesis notation [Jac89] to succinctly encode a single phylogenetic tree. They then encode further trees via edit operations (contraction and expansion of edges/bipartitions) based on the Robinson-Foulds distance [RF81]. This approach only allows for sequential access to the trees. However, we argue that random accesses are necessary to efficiently compute statistics for specific genomic regions of interest. Their approach also does not allow for a straight-forward memoization of recurring subtrees.

To the best of our knowledge, two existing phylogenetic tools use DAGs to memoize intermediate results: ASTRAL-III, a tool for inferring a species tree from a set of topologically distinct gene trees, represents and processes unique tip bipartitions only once [Zha+18]. Further, Larget [Lar13] employs memoization to compute the conditional clade probability only once per unique subtree. Gene Recombination Graphs (GRGs) [DPW24] encode the mutations of a collection of related genomes as a DAG in order to reuse computations. GRGs are conceptually similar to our bipartition-based DAG (Section 3.3.5) in that they partition the mutations of a set of genomes but model neither the evolutionary tree of the described genomes nor *where* on the evolutionary tree these mutations happened. Hence, GRGs do not support LCA queries and cannot encode mutations back to the ancestral state or multiple mutations along a single path from the root to a tip.

3.3 Design of the Genealogical Forest Data Structure

In recombining organisms such as humans (Section 2.1), the set of genealogical trees used to describe the evolutionary histories of different parts of the genome share common subtrees (a node including all of its descendants). Thus, we propose a data structure that encodes each subtree present in the input tree set exactly once, even if shared by multiple input trees (Figure 3.1.d). As tree branch lengths are irrelevant for the genetic variation based statistics and the LCA, we omit storing them.

We construct our data structure by contracting the set of unconnected trees into a DAG, where each root node represents a tree of the input tree set and each non-root node represents a unique subtree that is present in one or multiple tree(s) of the input tree set. Here, we consider each sample to be a subtree containing only itself. If two or more input trees share an identical subtree, the associated node in the DAG has multiple incoming edges; see for example `(B,C)` in Figure 3.1.d. These resulting DAGs are called *multitrees* [FZ94]. In them, each node, including all its direct and indirect descendants, induces a tree. Additionally, there is exactly one path from each root to each tip. In analogy to tree sequences (implemented in `tskit`), we call this data structure a *genealogical forest* and provide an open-source implementation called `gfkit`.[3] We choose this naming to highlight that the trees describe genealogies and use the established term "forest" to describe a collection of trees.

Moreover, we encode the (closely related) genomic sequences represented by the tips of the genealogical forest analogously to `tskit` (Figure 3.1.b). For this, we make the biologically realistic assumption that the evolutionary history of each genomic position is described by exactly one genealogical tree. However, note that several adjacent positions along the genome can and will often share the same tree. This assumption can potentially be relaxed – see future work (Section 3.8). For each site exhibiting genetic variation (Section 2.2), we store the ancestral state at the root and the respective mutations along the edges of the associated tree (Section 3.2). Note that each edge in a genealogical tree can be annotated by multiple mutations, as it describes the evolutionary history of multiple genomic sites.

[3] `https://github.com/lukashuebner/gfkit`; LGPL

Moreover, note that there might be multiple mutations per genomic site, including mutations back to the ancestral state and multiple mutations along a single path from the root to a tip (Figure 3.1.b).

3.3.1 Implications of Our Design

Each node in a genealogical forest represents a unique subtree present in one or more trees of the input set (Section 3.3). This allows queries using these trees' topologies or the associated genetic sequences to seamlessly reuse intermediate results on subtrees shared by *any* trees (*memoization*). In contrast, tree sequences allow for reusing intermediate results only among *adjacent* trees along the genome (Section 3.1.2). We implement this memoization by storing the query's intermediate results in a lookup table; using the ID of the respective DAG node as the key. We employ this memoization to accelerate the DAG post-order traversals that we utilize to compute the LCAs (Section 3.3.3) and allele frequencies (Section 3.3.4). From the allele frequencies we subsequently derive numerous statistics that are commonly used in population genetics (Section 2.4).

3.3.2 Computing the Number of Selected Samples in a Subtree

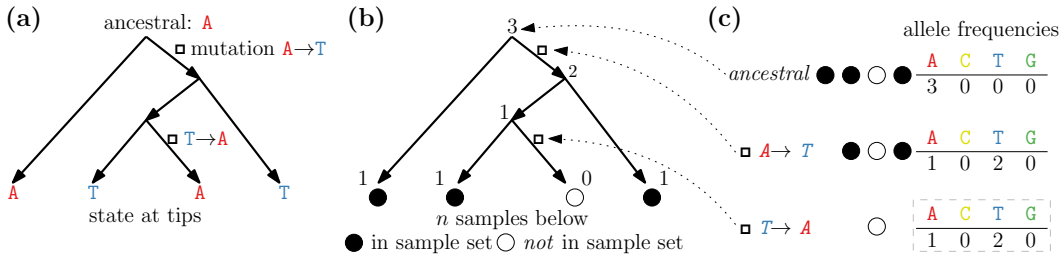


Figure 3.2 **(a)** Encoding a set of four sequences (here: one site) via a tree, an ancestral state, and mutations along the edges. **(b)** Queries take as input a subset of all samples, the *sample set*. We compute the number of samples in this sample set below each node using a post-order traversal. We need this information to **(c)** compute the allele frequencies (number of samples per genomic state) of each site by iterating over its mutations.

For each query to the genealogical forest data structure, the user selects a subset of the forest's samples to be considered (Section 2.4). Counting the number of selected samples which form part of each subtree of the genealogical forest (represented by a node in the DAG) is an important building block for computing allele frequencies and the LCA. For this, we define a post-order traversal on a DAG as iterating over all the tree's nodes such that all children of a given node are processed before the node itself. This allows us to reuse the intermediate result associated with these children using memoization (Section 3.3.1). Thus, to count the number of selected samples in each subtree, we assign $v(t) = 1$ for all tips t that represent samples selected in the user query and $v(t) = 0$ to all other tips. Next, we perform a post-order traversal on the DAG by assigning to each node n_k the sum over the values assigned to its children $\mathbb{C}(n_k)$, that is $v(n_k) = \sum_{n_c \in \mathbb{C}(n_k)} v(n_c)$ (Figure 3.2.a-b).

We store the edges E of this DAG sorted in an order that is compatible to a post-order traversal where we update the value of a node based on the values of its children. Thus, we can consider the value $v(n)$ of a node n as being finalized, once we processed all outgoing edges E^n_{out} of n. We ensure that the value associated with a node does not change anymore

after having been used by one of its parent nodes. For this, we sort all incoming edges of a node *after* its outgoing edges. That is, $e_i > e_o$ for all $e_i \in E^n_{\text{in}}$ and $e_o \in E^n_{\text{out}}$. Further, let $\mathbb{C}(n)$ be the children of n and $E^{\mathbb{C}(n)}_{\text{out}}$ the outgoing edges of all children of node n. We sort all edges in $E^{\mathbb{C}(n)}_{\text{out}}$ *before* the first incoming edge of n. That is, $e_{o,c} < e_{i,n}$ for all $e_{o,c} \in E^{\mathbb{C}(n)}_{\text{out}}$ and $e_{i,n} \in E^n_{\text{in}}$. This ensures that all values associated with a node's children have been finalized before we read them. Note that the construction algorithm outlined in Section 3.3.6 automatically outputs the genealogical forest's edges in this order.

We can thus perform a post-order traversal on the genealogical forest DAG via a single-pass linear scan over this edge list, keeping the values associated with each node in a hash map with the node ID as the key.

3.3.3 Computing the Lowest Common Ancestor

We compute the LCA (Section 2.4) of two or more tips in all trees represented by a genealogical forest using a variation of the algorithm described in Section 3.3.2. For this, we exploit the property that in a genealogical forest DAG, there exists exactly one path from each root to each tip. We chose this approach, as unrolling the DAG into a set of trees and answering LCA queries using the well-studied techniques for trees (e.g., Schieber and Vishkin [SV88]) would require memory linear in the number of nodes per tree times the number of trees. Further, using an LCA algorithm developed for general DAGs (e.g., Kowaluk and Lingas [KL05]) is asymptotically slower as it cannot exploit the above property.

First, we assign each tip t that was selected as part of the query the tuple $v(t) = (1, \varnothing)$ and each tip not selected the tuple $v(t) = (0, \varnothing)$. Here, the first element of the tuple counts the number of selected samples that form part of the subtree represented by the node. We accumulate this counter up the trees (bottom up from the tips) via a post-order traversal on the DAG. Moreover, if, for any node, this sample counter is equal to the number of selected samples for the first time, this node must be the LCA of this subtree. Thus, we store the ID of this node in the second element of the tuple and propagate this value up the tree, instead of the sample count. Note that we might select multiple nodes in the DAG as LCAs in a single query. However, as noted above, there exists exactly one path from each root to each tip. This ensures that for each root in the DAG (representing a genealogical tree), we pick *exactly one* node that is reachable from this root as the LCA.

As detailed in Section 3.3.2, we can perform a post-order traversal on the genealogical forest DAG in time linear in the number of edges in the DAG. In particular, the runtime does not depend on the number of samples in the input sample set. Therefore, computing the LCA for a large subset of the samples does not require more time than computing the LCA for two samples only (Section 3.6.2).

This algorithm could be extended to locate an almost-LCA – the lowest node under which "almost all" samples are located. This might increase the robustness of the biological interpretation against single "rogue" samples which have been erroneously included in the input sample set and cause the LCA to be substantially closer to the root than it would be without them.

3.3.4 Computing the Allele Frequencies and Derived Statistics

Most population genetic statistics depend on the allele frequencies [RTK20], that is, how many `A`s, `C`s, `T`s, and `G`s we observe at a given genomic site (Section 2.4). Therefore, we are only interested in variant sites, where at least two of these counts are greater than 0. Moreover, the computation of allele frequencies at different genomic sites are mathematically

independent of one another.[4] Given a query, including a selected subset of samples, we first count the number of these selected samples that are contained in each subtree (represented by the nodes in the genealogical forest DAG (Section 3.3.2)). Let $S = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$ be the set of alleles (here: possible genomic states) and $n(*)$ be the number of selected samples. For each site, we intend to compute the number of selected samples $n(x)$ which exhibit allele $x \in S$. First, we set all samples to be in the ancestral state, that is $n(x_{\text{ancestral}}) = n(*)$ and $n(x) = 0$ for all remaining x (Figure 3.2.c). Next, we iterate over the mutations at this site: Each mutation is associated with a subtree in the DAG and changes the state $x_i \in S$ of all samples contained in this subtree to the state $x_j \in S$. Thus, we successively decrement $n(x_i)$ and increment $n(x_j)$ by the number of selected samples contained in the respective subtree. From these allele frequencies, we can subsequently derive numerous common population genetics statistics (Section 2.4).

3.3.5 Memoizing on Shared Bipartitions

In contrast to computing the LCA, computing the allele frequencies is independent of the actual tree topologies as it only requires the sample bipartitions induced by the subtrees associated with the mutations. Remember, that these bipartitions separate all tips of a tree into two disjoint sets (Section 3.2). For example, two identical mutations in two distinct subtrees $((G_0,G_1),G_2)$ and $(G_0,(G_1,G_2))$ will identically affect the respective allele frequencies. Memoizing on shared *bipartitions* allows reusing (slightly) more intermediate results than memoizing on shared *subtrees* (Section 3.6.5). Next, we detail the construction of the regular (unique subtrees) genealogical forest and this variant (unique bipartitions).

3.3.6 Constructing the Genealogical Forest DAG

In order to construct the genealogical forest DAG, we assign unique IDs to all subtrees of the genealogical trees that form the input. For this, let h be a pseudorandom hash-function. First, we assign unique subtree IDs to the tips $T = \{t_0, t_1, \ldots, t_{|T|-1}\}$ of the tree: $\mathrm{id}(t_0) = h(0), \mathrm{id}(t_1) = h(1), \ldots, \mathrm{id}(t_{|T|-1}) = h(t_{|T|-1})$. Next, we compute the unique subtree ID of each non-tip node by applying the bit-wise exclusive or ($\oplus$) over all of its children (direct descendants), followed by hashing the result using h (Figure 3.3.a). That is, for a node n with children $\mathbb{C}(n)$, we compute $\mathrm{id}(n) = h(\oplus_{j \in \mathbb{C}(n)} \mathrm{id}(j))$. By using a symmetric function – that is, $\mathrm{id}(j) \oplus \mathrm{id}(i) = \mathrm{id}(i) \oplus \mathrm{id}(j)$ – we ensure that the order by which we process the children does not affect the resulting subtree ID. In order for the actual topology of the subtree to influence its ID, we break the linearity of the $\oplus$ operator via a hash function. Otherwise, the ID would solely be determined by the bipartition induced by the respective subtree. For example, the subtrees `((A,B)C)` and `(A(B,C))` would have the same subtree ID.

By using a 128 bit hash function, we ensure a collision probability of merely 10^{-16} when computing $\approx 3.8 \times 10^{11}$ hashes [BR05]. This corresponds approximately to the probability of a single bit-flip to occur at any given second in the 128 bit needed to output a single subtree ID [SPW09]. As this is negligible, we do not need to handle hash collisions.

As noted in Section 3.3.5, the allele frequencies are invariant w.r.t. the actual topology of a subtree, and solely determined by the samples selected for the query. For computing a unique *bipartition* ID, we therefore remove the linearity-breaking hash function h from the computation of the non-tip IDs, that is, $\mathrm{id}(n_i) = \oplus_{j \in \mathbb{C}(n)} \mathrm{id}(j)$ for any non-tip node n.

[4] Note that biological forces exists which cause the allele frequencies of adjacent sites to be correlated (*linkage*) [Coo+10].

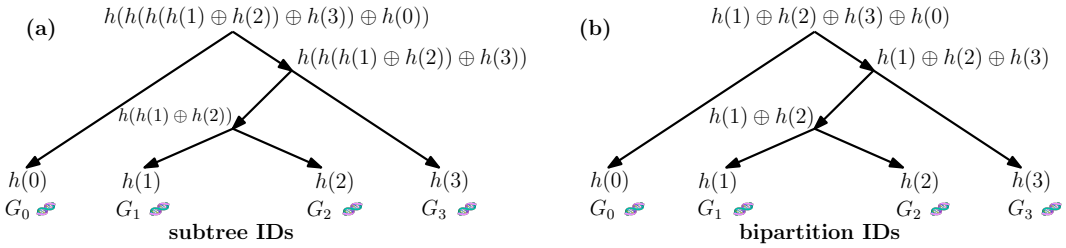


Figure 3.3 (a) Four sample genomes mapped to $t_i \in \{0, 1, 2, 3\}$. The unique subtree IDs of the tree's tips (i.e., samples) are $h(t_i)$, where h is a pseudorandom hash function. For each non-tip node, we compute the unique subtree IDs by applying h to the bit-wise exclusive or ($\oplus$) of all of its children. Thus, the resulting ID uniquely identifies a *subtree* including its topology. **(b)** Not breaking the linearity of the bit-wise $\oplus$ results in a unique ID per set of samples in a subtree (that is, not including its topology; called *bipartition*).

Construction Algorithm We construct the genealogical forest DAG from a given tree set by iterating over the input trees[5] and processing the nodes of each tree in post-order. For each tree node (representing the subtree induced by it and its descendants) encountered during this post-order traversal, we compute the unique subtree ID and add the subtree to the DAG if it is not already present. For the sake of simplicity, we make one exception to this rule and ensure that there exists exactly one distinct root in the DAG for each tree in the input tree set. That is, even if two trees are topologically exactly identical, each is represented by a separate root node in the DAG and has thus a unique tree ID. The trees evidently still share all subtrees below their respective, distinct, root node.

Recent versions of `tskit` provide information on which edges have been inserted or removed when moving from one tree to the next along the genome. We exploit this information in order to only recompute those subtree IDs that might have changed, that is, all subtrees induced by nodes that are on the path from a changed edge to the tree root. Utilizing this information yields speedups between 2 to 5, depending on the dataset (data not shown).

Analogously to `tskit`, we encode the genomic sequence by storing the ancestral state as well as the respective mutations annotated at *nodes* of the DAG for each genomic site.

3.3.7 Balanced-Parenthesis Encoding of a Forest

Instead of encoding a genealogical forest as a DAG by means of explicit nodes and edges, a representative extending the balanced parenthesis encoding for trees [Jac89] with back-references is possible. This encoding is space-efficient and a post-order traversal over a balanced parenthesis encoded tree is a simple linear scan. Exploring the full design space of this approach and providing an efficient implementation represents an interesting challenge. However, initial experiments did not yield substantial speedups, and hence we do not include them here.

3.4 Experimental Setup

We implement our algorithms in C++20 and build our tool using CMake 3.25.1, GCC 12.1, and `ld` 2.38. We execute our experiments on an AMD EPYC 7551P processor running at

[5] The order of trees in not relevant.

Table 3.1 Tree sequence datasets used in benchmarks.

Collection	Chr.	Format	Authors	Link
SGDP	all	`tskit`	[Woh+22]	`https://doi.org/10.5281/zenodo.3052359`
SGDP	all	`gfkit`	us	`https://doi.org/10.5281/zenodo.11241730`
TGP	all	`tskit`	[Woh+22]	`https://doi.org/10.5281/zenodo.3051855`
TGP	all	`gfkit`	us	`https://doi.org/10.5281/zenodo.11241619`
Unified	all	`tskit`	[Woh+22]	`https://doi.org/10.5281/zenodo.5495535`
Unified	all	`gfkit`	us	`https://doi.org/10.5281/zenodo.11241788`
Sim. 640k	20	`tskit`	us	`https://doi.org/10.5281/zenodo.11241938`
Sim. 640k	20	`gfkit`	us	`https://doi.org/10.5281/zenodo.11241938`

2 GHz with 64 MiB of shared L3, 512 KiB of core-local L2, 32 KiB of core-local L1 data, and 64 KiB core-local L1 instruction cache. We use 8 banks of 32 GiB DDR4 RAM running at 2667 MT s^{-1}. As our experiments are single threaded, only a single socket is being used. Further, we disable the CPU's frequency scaling and employ thread pinning. We compare `tskit` git rev `77faade5` and `gfkit` version `fbd2740`.[6] The tree sequence datasets (Table 3.1) were inferred using `tsinfer` [Kel+19] version `0.2.1` and dated using `tsdate` [Woh+22] version `0.1.4`.[7]

3.5 Datasets

We evaluate our method on three freely-available empirical human (*Homo sapiens*; GRCh38) tree sequence collections (Table 3.1): Thousand Genomes Project (TGP) [Aut+15] phase 3 autosomes, Simons Genome Diversity Project (SGDP) [Mal+16] autosomes, and the collection inferred by Wohns *et al.* [Woh+22] ("Unified"). We use all 22 autosomes for each of these collections in our experiments. We choose these specific tree sequence collections because to the best of our knowledge, they constitute the sole publicly-available empirical collections.

In addition, we simulate a human dataset containing 640 000 samples. As we do not observe substantial runtime differences between distinct chromosomes of the empirical genomic data collections, we limit our simulated data benchmarks to chromosome 20 to conserve computational resources and reduce our CO_2 footprint. We choose this approach analogously to Wohns *et al.* [Woh+22], who use chromosome 20, as they consider it as being representative of genome-wide patterns. We simulate the "Sim. 640k" dataset (Table 3.1) containing 640 000 samples using `stdpopsim 0.2.0`[8] and the `HapMapII_GRCh38` genetic map. We convert the respective tree sequences to genealogical forest files via `gfkit` version `fbd2740`.

3.6 Evaluation

We compare our proposed data structure genealogical forests (implemented in `gfkit`) regarding query speed (Sections 3.6.1 and 3.6.2) and storage space used (Section 3.6.6) against the state-of-the-art implementation tree sequences (`tskit`). Moreover, we assess the algorithmic

[6] `https://github.com/tskit-dev/tskit` and `https://github.com/lukashuebner/gfkit`
[7] The description on Zenodo is incorrect; see: `https://tskit-dev.slack.com/archives/C010834D669/p1713340040404519`
[8] `https://github.com/popsim-consortium/stdpopsim`

reasons for the obtained speedups (Sections 3.6.3 and 3.6.4) and report the time required to convert the `tskit` tree sequence format into the `gfkit` data structure (Section 3.6.7).

3.6.1 Speedup for Computing Statistics Based on the Allele Frequencies

We evaluate the runtime of our proposed genealogical forest data structure and its associated implementation `gfkit` by comparing it to the state-of-the-art reference implementation, `tskit`. We benchmark the runtimes of various important statistics in population genetics (Section 2.4), including statistics that are based on the allele frequencies (e.g., sequence diversity) as well as on the topology (e.g., LCA). We use 10 repeats for each runtime measurement on each of the 22 autosomal chromosomes of each empirical collection (TGP, SGDP, and Unified) and on chromosome 20 of the simulated dataset with 640 000 samples (Section 3.5). The mean standard deviation of runtimes across the 10 repeats of each statistic, collection, and chromosome is below 1.3 %. We report a median speedup of 4.0 of `gfkit` (ours) over `tskit` (state-of-the-art) for computing various allele frequency-based statistics (Figure 3.4) and pairwise LCA queries. The absolute runtimes range from 775 to 1770 ms for `tskit` and 152 to 340 ms for `gfkit`.

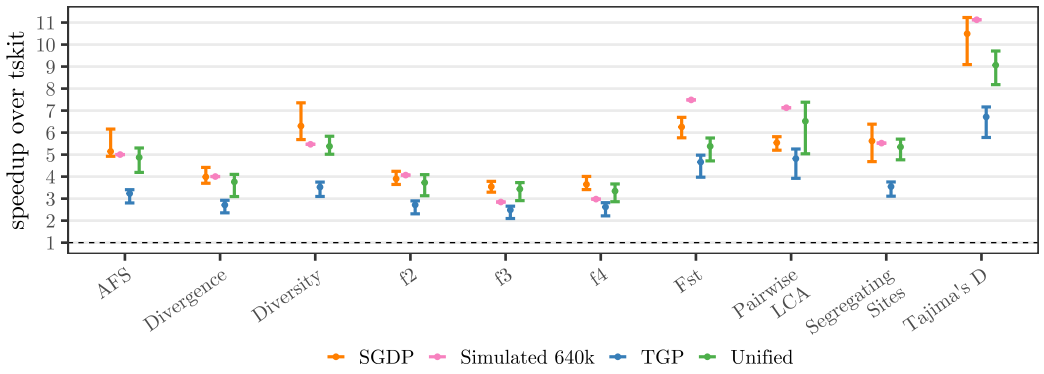


Figure 3.4 Speedup of `gfkit` (ours) over `tskit` (state-of-the-art) for computing statistics on three empirical and one simulated dataset (human). Bars indicate the speedup range and dots the median speedup across all chromosomes of the respective collection. We use all 22 autosomal chromosomes from the Thousand Genomes Project (TGP), Simons Genome Diversity Project (SGDP), and the Unified collection. We use chromosome 20 of a simulated human collection containing 640 000 samples. AFS: Allele Frequency Spectrum. f{2,3,4}: Patterson's f. Fst: Fixation Index.

Profiling via the Intel VTune tool shows that `gfkit` spends over 90 % of its runtime in the post-order traversal during computation of the allele frequencies. The numerical computations for the actual statistics thus account for less than 10 % of the runtime. Therefore, the number of nodes in the genealogical forest DAG, as well as the in-degree of those nodes, appear to constitute the dominating runtime factors. These values correspond to the number of unique subtrees in the genealogical trees used to construct the genealogical forest, and the number of times that an intermediate result can be reused during the post-order traversal, respectively. Thus, we provide more details on these measurements in Sections 3.6.3 and 3.6.4.

3.6.2 Speedup for Computing the Lowest Common Ancestor

We compare `gfkit`'s and `tskit`'s runtimes for computing the LCA of two ("pairwise") or more samples. The asymptotic runtime of `gfkit`'s LCA-algorithm does not depend on the

number of selected samples (Section 3.3.3). In contrast, given three or more samples, `tskit` performs an analogous number of pair-wise LCA-queries: First, `tskit` computes the LCA of two arbitrary samples, which it subsequently uses as input for the next LCA query; together with another sample from the input. Thereby, after processing all samples, `tskit` obtains the LCA over all samples that were selected in the input. Hence, `tskit`'s LCA-algorithm scales linearly with the number of samples in the input as well as tree count and size.

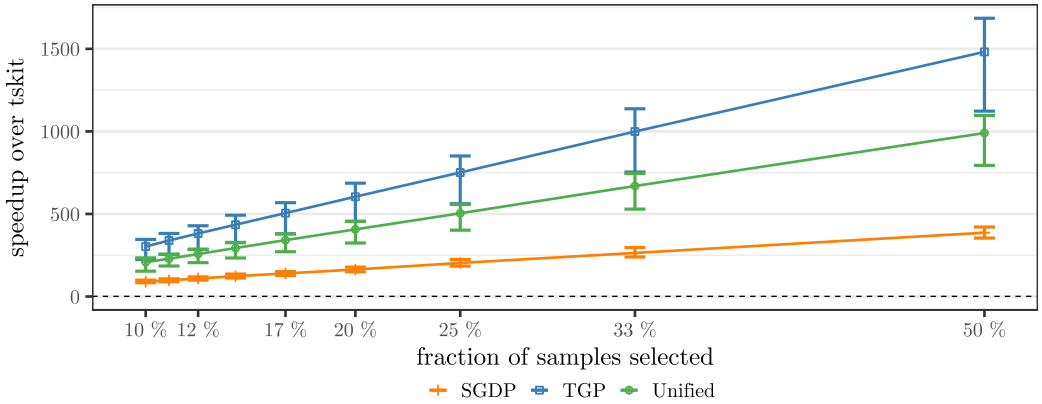


Figure 3.5 Speedup of `gfkit` (ours) over `tskit` (state-of-the-art) for computing the Lowest Common Ancestor (LCA) of every n-th sample in the dataset. The runtime of `gfkit`'s LCA-algorithm does not depend on the number of samples in the sample set. In contrast, the runtime of `tskit`'s LCA-algorithm depends linearly on the number samples in the sample set. Note the log-scaling of the y-axis. Experiments on the simulated dataset took 64 min for `gfkit` (median 155 ms per query) but did not finish for `tskit` within a week.

We report the speedups of `gfkit` over `tskit` when computing the LCA (Figures 3.4 and 3.5). In order to save computational resources and reduce our environmental footprint, we perform only 3 repeats when computing the LCA of more than two samples using `tskit` (runtime up to 34 min). We observe speedups of `gfkit` over `tskit` of 5.5 (median) for pairwise queries, 208 (median) when selecting 10 % of the samples, and 990 (median) when selecting 50 % of the samples as input sample set. All samples were selected uniformly at random.

3.6.3 Proportion of Subtrees that are Unique

Each query spends the majority of its time performing the post-order traversal. Thus, the number of edges and nodes in the `gfkit` DAG is *the* determining runtime factor, with the number of nodes in the DAG being (nearly[9]) equal to number of unique subtrees in the input tree set. We report on the number of overall subtrees in the input as well as the absolute and relative number of unique subtrees (Table 3.2). For example, in the TGP collection, 0.48 % (mean) of the subtrees are unique per chromosome. Additionally, we report that the absolute number of unique subtrees in a simulated dataset (human, chromosome 20) with 640 000 samples is not substantially higher than for the shown empirical collections containing 554 to 7508 samples, respectively. Overall, only 0.002 to 2 % of the subtrees in the tested collections are unique. This proportion decreases as we include more samples (and

[9] If multiple trees are exactly identical, we add a distinct root for each of them (Section 3.3.6).

thus subtrees) from the same species. This is because the absolute number of unique subtrees does not increase substantially for larger collections, thus, neither does the runtime of the queries. The number of mutations per subtree is 3 to 4 orders of magnitude smaller[10] for the 640 000-sample dataset compared to the collections with $\leq$ 7508 samples. Therefore, there is less evolutionary signal to resolve the evolutionary history of the samples, possibly inducing larger unresolved subtrees. It remains an open question how other data characteristics, for example the species when not considering human populations, influence the number of unique subtrees.

Table 3.2 Number of overall and unique (i.e, distinct) subtrees and the proportion of subtrees that are unique. The ranges given cover all 22 autosomal chromosomes of the respective collection. We report the arithmetic mean and standard deviation across all chromosomes of a collection.

Collection	Chr.	Samples	Trees $\times 10^6$	Subtrees $\times 10^6$	Uniq. Subtrees $\times 10^6$	Uniq. Subtrees / Subtrees
SGDP	all	554	0.1 to 0.7	84 to 587	1.8 to 11	0.020 ± 0.001
TGP	all	5008	0.3 to 2.1	2150 to 14 916	11 to 69	0.0048 ± 0.0003
Unified	all	7508	0.04 to 0.2	698 to 2708	3.2 to 18	0.006 ± 0.002
Sim. 640k	20	640 000	0.5	695 185	14	0.000 02

3.6.4 Reusing Shared Subtrees and Intermediate Results

Apart from the number of nodes in the `gfkit` DAG, the number of edges also influences the runtime of the post-order traversal. Moreover, the proportion of edges to nodes can also serve as an indicator for memoization performance. This is because, during the post-order traversal, the in-degree of a node is equal to the number of times its result is used (and reused). We report that each intermediate result on a unique subtree is being reused on average 4.12 (SGDP), 5.49 (TGP), 5.10 (Unified), or 1.99 (Simulated 640k) times.

Table 3.3 Average in-degree of non-root nodes in the `gfkit` DAG. The in-degree of a node in the DAG is equal to the number of times its intermediate result is reused during the post-order traversal.

Dataset	Chr.	Mean In-degree Bipartition	Mean In-degree Subtree
SGDP	all	4.44	4.12
TGP	all	5.89	5.49
Unified	all	5.20	5.10
Sim. 640k	20	—	1.99

3.6.5 Speedups when Memoizing on Shared Bipartitions

We also benchmark queries on the bipartition-DAG described in Section 3.3.5. Here, we observe a median speedup of 4.7 over `tskit` across all empirical collections (Figure 3.6). Queries on the bipartition-DAG are on average 1.14 ± 0.09 (mean $\pm$ standard deviation)

[10] Mutations per subtree: $\approx$ Sim. 640k: 0.000 000 7, SGDP: 0.002, TGP: 0.0002, and Unified: 0.003

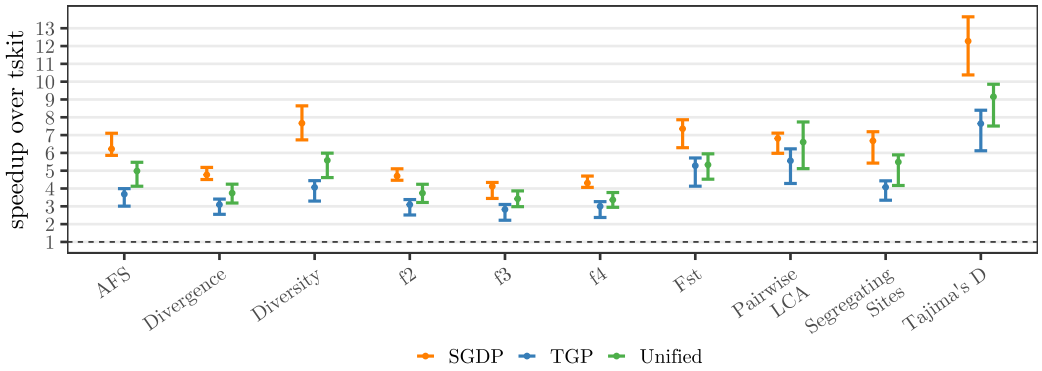


Figure 3.6 Speedup of `gfkit` (ours) using a *bipartition*-DAG over `tskit` (state-of-the-art) when computing allele frequency based statistics on three empirical and one simulated dataset (human). The bars indicate the speedup range and the dot the median speedup across all chromosomes of the respective collection. We use all 22 autosomal chromosomes for TGP, SGDP, and the Unified collection. We omit the simulated human dataset with 640 000 samples to save on computational resources.

times faster than on the subtree-DAG. There are 5 to 20 % fewer unique bipartitions than there are unique subtrees; corresponding to fewer nodes in the bipartition-DAG. The average in-degree in the bipartition-DAG, and thus number of times we reuse an intermediate result, ranges from 4.44 to 5.89, compared to 4.12 to 5.49 on the subtree-DAG (Table 3.3). These two measurements explain the (moderately) faster runtimes of the bipartition-DAG compared to the subtree-DAG. However, as the bipartition-DAG does not support topology-aware queries (e.g., LCA), we do not consider this tradeoff worthwhile, taking into account the increased code complexity and storage, unless additional optimizations further reduce these runtimes.

3.6.6 Storage Space Needed for Encoding the Forest

Conceptually, a genealogical forest DAG factors-out all unique subtrees that a tree sequence describes using its edge insertions and removals. Concerning the space usage, there are two contrary effects: On the one hand, tree sequences store some identical subtrees multiple times, but using distinct edges and/or nodes. Here, a genealogical forest constitutes the more space efficient representation. On the other hand, a single edge insertion or removal can induce multiple new subtrees along the path from the insertion or removal point up to the tree root. Here, tree sequences are the more space efficient representation.

In order to quantify the trade-off between the two effects, we compare the size of the genealogical forest DAG against the size of the respective tree sequence. However, the tree sequence implementation `tskit` stores a variety of additional meta-data, to support operations which we currently do not implement in `gfkit`. We thus perform a theoretical, instead of an empirical space requirement comparison.

Both, tree sequences and genealogical forests could minimally be described using their (directed) edges and sequence information. Let ι be the number of bits required to encode a node ID and let ϕ be the number of bits required to encode a position in the genome. Thus, choosing an edge list (see below) we require $2 \cdot \iota$ bit for storing each edge in a genealogical forest. In a tree sequence, an edge is valid only for a specified region of the genome, requiring $2 \cdot \phi$ additional bits per edge in order to store this information. Note, however, that the

number of nodes and edges differs between `tskit` and `gfkit` and these numbers are thus not directly comparable. Additionally, we cannot arbitrarily sort `tskit`'s edges without incurring a performance penalty, as we require the insertion order to be able to efficiently move from one tree to the next along the genome.[11] In `gfkit`, all outgoing edges of a node could be stored consecutively in an edge list, which would allow omitting some `from`-node IDs, conceptually creating an adjacency array.

Both, `tskit` and `gfkit`, describe the genomic sequences using one ancestral state and a set of mutations per genome site. We can encode each mutation using a site ID, the node ID at which the mutation occurs, and the derived state. Let s be the number of sites, σ be the number of bits per site ID, γ be the number of bits per genomic state, and m be the overall number of mutations. The sequence information thus (theoretically) requires $s \cdot \gamma + m \cdot (\sigma + \phi + \gamma)$ bit. In practice, `tskit` and `gfkit` store additional information: For example, in order to avoid traversing up the tree to the parent mutation (or root node), `tskit` stores a pointer to the parent mutation (64 bit) and `gfkit` stores the parent mutation's state (2 bit). However, we omit these and other implementation-specific constants here.

Table 3.4 Theoretical space usage of `tskit` vs `gfkit`. The description of the trees is substantially larger than the description of the sequence (ancestral states and mutations). `tskit` encodes trees as a tree sequence (Section 3.1.2), whereas `gfkit` encodes them via a genealogical forest (Section 3.3).

Collection	Chr.	`tskit`	`gfkit`	Sequence
SGDP	all	1076 MiB	3891 MiB	123 MiB
TGP	all	4577 MiB	32 134 MiB	310 MiB
Unified	all	1684 MiB	7843 MiB	874 MiB
Sim. 640k	20	53 MiB	198 MiB	3.74 MiB

For our calculations (Table 3.4), we assume $\iota = \phi = \sigma = 32$ bit integers for node IDs, site IDs, and genomic positions[12], and four possible genomic states, thus $\gamma = 2$ bit. We conclude that the edge removals and insertions used to store the set of genealogical trees by `tskit` are more-space efficient than `gfkit`'s method of storing a node for each unique subtree. However, we argue that, as the differences are in the same order of magnitude as implementation-specific constants (e.g., choice of data-types or explicitly storing IDs), one could mitigate this effect. Therefore, it remains a tradeoff between query-speed and space usage. It is subject of future work to evaluate the genealogical forest variants using the compressed edge list (above) and the balanced parenthesis representation (Section 3.3.7).

3.6.7 Converting Tree Sequences to Genealogical Forests

To the best of our knowledge, there is no fundamental reason why tools could not output genealogical forests instead of tree sequences once we include support for storing coalescent times (Section 3.8). Currently, however, existing tools output the established `tskit` format – and will probably continue doing so for a while, since the current implementation of our data structure, `gfkit`, does not support all features of `tskit` yet. While we provide corresponding `gfkit` files (Section 3.5) for the openly available `tskit` collections (Section 3.5), the time

[11] We could determine the deletion order on the fly by using a priority queue containing only the currently active edges.

[12] This is true for `gfkit`. However, `tskit` uses $\phi = 64$ bit IEEE 754 floating-point values for genomic positions.

required to convert `tskit` to `gfkit` input files is not negligible. Kelleher *et al.* [Kel+19] report tree sequence inference times of 5 min for chromosome 20 of SGDP and 2 h for chromosome 20 of TGP using 40 cores. Hengstler [Hen24] reports reducing these times by half using a highly-tuned non-feature-complete implementation. However, we are able to convert output files from the `tskit` to the `gfkit` format on a single core in 6.5 s and 123 s respectively. We thus argue that one can convert empirical collections in a negligible amount of time compared to the time required for inferring them.

3.7 Numerical Stability

Basic arithmetic operations on IEEE 754 [IEEE754] floating-point numbers, such as additions or multiplications, always induce rounding errors [Gol91] (Section 2.10). The magnitude of these errors directly depends on the difference in the order of magnitude between the two operands. Thus, computing running sums, where we sum over many small values to an ever-growing sum, is particular rounding-error prone. We often compute statistics in population genetics on a per-site basis (Section 2.4) and then average these over all sites. Using a running sum, as currently implemented in `tskit` and `gfkit`, leads to increasing round-off errors, the more variant genetic sites there are. Some statistics, like Patterson's f_4 entail multiplying the number of samples in four different sample sets. With more than 262 143 samples,[13] the product might exceed the size of a 64 bit integer. We (for the sake of result verification), as well as `tskit`, thus choose to represent these products as 64 bit IEEE 754 floating-point numbers, potentially introducing substantial numerical errors.

It is currently unknown if the errors introduced by these simplifications influence the biological interpretation of the results. Further, the existence of rounding errors in the intermediate and end result implies that the operation order influences their exact value, even though the input operands are bit-identical, thus impacting reproducibility. Even when the software version and the random seeds are fixed, different algorithms, compiler optimizations, or degrees of parallelism influence the order of operations (Section 2.10). Reports on these effects influencing the results of scientific computations exist for example in the fields of phylogenetics [DFS18], sheet metal forming [Die12], and fluid simulations [Wie+19].

An analysis of the impact of numerical errors on population genetics statistics, their downstream analyses, and result reproducibility is urgently required. Ideally, a standardized order of calculations would lead to all software in the field outputting comparable and reproducible results with well-understood numerical inaccuracies. Further, results should be invariant under addition of further – but ignored in the query at hand – genomes and mutations to the dataset. In our case, the numerical calculations require less than 10 % of overall runtime. Hence, we are optimistic that introducing arbitrary precision calculations might have negligible runtime overhead.

3.8 Conclusion and Future Work

In recombining organisms, a set of genealogical trees is often used to describe the evolutionary history of the studied samples. While tree sequences (`tskit`) compress these trees using edit operations from one tree to the next along the genome, genealogical forest (`gfkit`) compresses them into a DAG where each node represents a unique subtree of the input (Section 3.3). While the genealogical forest encoding (theoretically) requires a factor 3.6 to 7.0 more space

[13] When the four sample sets have the same size: $(262144/4)^4 = 2^{64}$

than tree sequences (Section 3.6.6), it also yields speedups by a factor of 2.1 to 11.2 (median 4.0) when computing pairwise LCA queries and important statistics in population genetics (e.g., sequence diversity). In contrast to `tskit`'s LCA-algorithm, the runtime of `gfkit`'s LCA-algorithm does not depend on the number of samples selected in the query. Thus, `gfkit`'s LCA queries are substantially faster, for instance, a speedup of 208 (median) when querying 10 % of samples and a speedup of 990 (median) when querying 50 % of samples. Additionally, we describe, implement, and benchmark an alternative data structure, which represents the set of genealogical trees as a DAG in which each node represents a unique bipartition in the input. This variant is slightly faster but does not support topology-aware queries (e.g., LCA). To explain this, we show, that subtrees containing the same samples but having a different topology are indeed infrequent in the analyzed datasets.

As more genomes are being sequenced, future datasets will contain a multiple of today's samples. However, we observe speedups (and runtimes) comparable to the largest existing empirical datasets for a simulated human dataset with 640 000 samples (Section 3.6.1). We show, that this might be due to the number of unique subtrees being comparable to those of current empirical datasets with at most 7508 samples, thus resulting in a DAG of about the same size. As the post-order traversal on the genealogical forest DAG consumes over 90 % of a query's runtime, the size of this DAG is the determining runtime factor. Which characteristics of the input influence the number of unique subtrees remains an open question.

`Tskit` was not build in a day and neither was `gfkit`. Currently, `gfkit` does not support all of `tskit`'s features, yet most of these should be straight-forward to implement. For example, `gfkit` currently only supports sample weights of 0 and 1 (a sample is in the sample set, or it is not), as these are sufficient for implementing the allele frequencies and derived statistics as well as LCA queries. Further, `gfkit` currently only supports site statistics on the entire genome, that is, neither branch- nor node-based statistics as well as no sliding windows over the genomic sites. We plan on adding support for storing coalescent times at inner nodes, enabling branch-length based statistics and the application of LCAs described in Section 2.4. Additionally, `gfkit` assumes that for each genomic site, a single tree describes the evolutionary history of all tips, while `tskit` allows for multiple (partial) trees.

Tree sequences are an instantiation of Ancestral Recombination Graphs and can be augmented with nodes representing recombination events between two individuals [Won+23b]. In a genealogical forest, each tree describes the evolutionary history of *a part of the* genome. Thus, the full genetic code of an individual is described by a subset of nodes. We can annotate a recombination event between these subsets. However, this is not implemented yet.

Next to lifting these limitations, additional improvements include evaluating a top-tree based genealogical forest (Section 3.2) and extending LCA queries with more than two samples to also report ancestors which are common to "almost all" selected samples. This will yield the query robust against single samples that have been erroneously included in the query (Section 3.3.3).

Tree sequences opened up new possibilities for storing and processing large sets of genealogical trees and sequences in population genetics. Genealogical forests provide a substantial reduction in runtime for site-based statistics and LCA calculations over the state-of-the-art. Additionally, they render developing efficient algorithms more intuitive as these algorithms can be implemented, for example, via post-order traversals that allow to automatically reuse intermediate results. We believe that these improvements will boost the development of new analyses in the field of population genetics. For instance, they might be used to develop appropriate optimization criteria and techniques for automatically determining subpopulations.

4 ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms

Attribution: This chapter is based on the peer-reviewed publication "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms" [Hüb+22a], on which I share first authorship with Demian Hespe. Peter Sanders and Alexandros Stamatakis contributed as advisors. Large parts of this chapter have been copied either in modified or verbatim form from this publication [Hüb+22a] or the corresponding technical report [Hüb+22b]. I designed the core of ReStore's data distribution and algorithms in close collaboration with Demian Hespe. During the implementation, I focused on the data distribution ("creating a checkpoint") while Demian Hespe focused on data recovery after a failure. Moreover, I conducted and evaluated all experiments, designed the method to recover lost replicas, provided the theoretical analysis of the irrecoverable-data-loss probability, implemented the serialization interface, and conducted the related-work reproducibility study.

Summary: The number of CPU cores in High Performance Computing (HPC) systems increases, and existing petascale systems already fail 0.8 *to* 5.7 *times per day. As future exascale systems are expected to experience hardware failures every* 30 *to* 60 *minutes, mitigating hardware failures constitutes a major challenge in HPC.*
It is impractical for applications to either, request replacement for failed CPU cores after a failure, or to maintain spares. Applications must thus continue with the remaining cores. During failure-recovery, algorithms have to restore lost data that resided on the failed cores, as well as re-distribute the workload and respective parts of the input data to the remaining cores (shrinking recovery). However, current checkpointing libraries often do not support the data redistribution required for shrinking recovery and thus necessitates replacement cores. Further, they typically write checkpoints to the parallel file system instead of the main memory, which incurs slow recoveries due to low disk access speeds and disk congestion.
We present an algorithmic framework and its C++ library implementation ReStore, which enables scalable, shrinking data-recovery to mitigate hardware failures in MPI programs. We store all required data in memory via an appropriate data distribution and replication strategy, yielding substantially faster recovery than under standard checkpointing schemes, which rely on a parallel file system. ReStore allows the application developer to specify which data to load on which core, and thus supports the data redistribution required by shrinking recovery as well as recovery using spare compute nodes. We evaluate ReStore in both controlled, isolated environments and real-world applications.
Our experiments show loading times of lost data in the range of milliseconds on up to 24 576 *cores and a substantial speedup of the recovery time for the failure-tolerant version of the widely used bioinformatics tool RAxML-NG. To the best of our*

knowledge, ReStore is the first in-memory checkpointing library that supports shrinking recovery. Therefore, ReStore enables scientific applications to mitigate from hardware failures in milliseconds, requiring neither replacement nor spare resources.

4.1 Introduction

Next to work-efficient data structures and algorithms (Chapter 3), the data avalanche in evolutionary bioinformatics also necessitates increased parallelism in our applications. However, with an increasing number of CPU cores (here, Processing Elements (PEs)), the probability that a program experiences a hardware failure during its execution increases. Current petascale systems already fail 0.8 to 5.7 times per day, while future exascale systems are expected to experience hardware failures every 30 to 60 minutes [Cap+14; DHR15; Sni+14] (Section 2.7). Therefore, mitigating these hardware failures constitutes a major challenge for future exascale systems [SDM10].

As discussed in Section 2.8, the prevalent [Her+15, ch. 1] failure-mitigation strategy in scientific computing on High Performance Computing (HPC) system is checkpoint/restart. Checkpoint/restart strategies can be further classified into application-level versus system-level libraries, if checkpoints are coordinated/uncoordinated among the PE, and if checkpoints are created in-memory or written to the parallel file system (PFS; Section 2.7). As discussed in Section 2.7, application-level checkpoints allow the application to specify which sections of its memory shall be backed-up, drastically reducing the amount of backed-up data over system-level checkpoints [Hüb+21b]. In coordinated checkpointing strategies, all PEs write their checkpoints at the same time, and are thus easier to implement than uncoordinated checkpointing schemes (Section 2.8). As synchronization is cheap in modern HPC systems with high-bandwidth, low-latency interconnections, Gavaskar and Subbarao [GS13] recommend coordinated checkpointing when developing applications for these systems. Hence, we focus on coordinated, application-level checkpointing schemes in this chapter.

Many existing checkpointing libraries [Aga+04; Bar+17; Bau+11; BH14; Gam+14; Lu05; Moo+10; Nic+19; Sha+19; TH14] only target recovery using the *substitute* strategy, where a replacement PE takes over the work previously performed by the failed PEs (Section 2.7). While this circumvents the need for re-balancing the workload and simplifies (re-)loading the required data, reserving idle PE for this purpose constitutes a waste of resources. In contrast, when employing the *shrink* strategy, the program's load balancer (re-)distributes the work performed by the failed PE among the remaining, or *surviving*, PEs (Section 2.7). This often causes all PEs that where assigned a new work package to load the respective parts of the static (input) data; often from the PFS, thus causing congestion. In previous work [Hüb+21b], we observe this bottleneck during recovery in a fault-tolerant phylogenetic tree search (Section 2.3). Note that this partial loading of data requires the data fractions stored in a checkpoint to be individually addressable, using, for example, explicit IDs.

In summary, while shrinking recovery does not necessitate spare PEs, it reloads fractions of the data on many or even all PEs during recovery. Thus, the respective checkpointing library has to support this operation without inducing congestion on the storage medium.

4.1.1 Contribution

We introduce ReStore, an in-memory checkpointing library, which supports recovery using both, the substitute, and the shrink strategy. This allows all available PEs to participate in the computation at all times, instead of reserving some as spares to replace failed PEs. Particularly, ReStore supports the partial load operations required for this shrinking recovery

via an involved strategic data distribution and recovery mechanism. Keeping the checkpoints in memory avoids the bottlenecks involved in a PFS and increases scalability. Further, the user can fine-tune ReStore to favor faster checkpoint creation or faster data recovery by explicitly choosing an appropriate data distribution strategy. Rapid recovery is particularly important for data that the program never or rarely changes but has to be redistributed after every failure, for example, the input data.

To the best of our knowledge, ReStore is the first general purpose checkpointing solution that allows for in-memory recovery without requiring spare PEs. We show the real-world applicability of ReStore by integrating it into a fault-tolerant version of the widely-used phylogenetic inference tool RAxML-NG and a distributed k-means implementation. Since its publication, ReStore has been used for creating (compressed) checkpoints in a fault-tolerant PageRank [Pag+99] implementation and will continue to facilitate the development of fault-tolerant algorithms by providing rapid and scalable checkpoint creation and recovery.

4.1.2 Outline of this Chapter

The remainder of this chapter is structured as follows: In Section 4.2, we first review important concepts. Next, we offer an overview of existing checkpointing libraries (Section 4.3). Further, we discuss the design of ReStore's replication and data distribution scheme (Sections 4.4.1 and 4.4.2), and analyze ReStore's memory usage (Section 4.4.3) as well as the probability of irrecoverable data loss (Section 4.4.4). We also present alternative data-distributions, which enable the recovery of lost replicas after a PE failure (Section 4.4.5). After discussing our implementation of ReStore in Section 4.5, we experimentally evaluate it via isolated measurements (Section 4.6.2), integrate it into real-world applications (Section 4.6.3), and compare our results to other checkpointing libraries and recovering data from the PFS (Section 4.6.4). Finally, we conclude in Section 4.7.

4.2 Preliminaries

As discussed in Section 2.5, in distributed-memory parallel programs using the *Message Passing Interface* (*MPI*; Section 2.6), p processes (or *Processing Elements (PEs)*) run on multiple machines (or compute *nodes*) and communicate via messages exchanged over the network. We consider two important factors for evaluating the running time of such parallel algorithms: The *bottleneck* number of messages sent and received, and the *bottleneck communication volume* (details in Section 2.5). As failures, we consider the case that one or more PEs suddenly stop working and do not contribute to the computation anymore – which we will refer to as *failed* (Section 2.7). The upcoming MPI standard 5.0 will support mechanisms for mitigating failures of compute nodes and/or network components (Section 2.9). Currently, there exist two actively developed MPI implementations which already support MPI's failure-mitigation mechanisms: MPICH [Gro02] as of version 3.1 and User Level Failure Mitigation (ULFM) [Bla+13], which has now been merged into OpenMPI 5.

4.3 Related Work

Scientific applications are increasingly implemented/extended to be able to mitigate hardware failures. Examples include a numeric linear equation and partial equation solver [Ali+16], a plasma simulation [Obe+17], a molecular dynamics simulation [Lag+16], a Fast Fourier Transformation [EG03], and an algorithm for phylogenetic inference [Hüb+21b].

Table 4.1 Comparison of checkpointing libraries. Note that ftRMA [BH14] is Cray-only. The author-provided examples for Fenix [Gam+14] segfault on our system. `GPI_CP` [Bar+17] requires `libiverbs`, GPI-2, and password-less ssh-login on all compute nodes. [1] The program needs to allocate spare nodes, which participate in the computation only in case of a failure. [2] The program needs to allocate spare nodes and, in addition, further nodes exclusively used for storing checkpoints.

	ftRMA [BH14]	**Fenix** [Gam+14]	**SCR** [Moo+10]	**Lu** [Lu05]	**GPI_CP** [Bar+17]	***ReStore***
Features						
in-memory checkpointing	✓	✓	✗	✓	✓	✓
substituting recovery	✓	✓	✓	✓	✓	✓
shrinking recovery	✗	✗	✗	✗	✗	✓
all nodes participate in computation	✗[2]	(✓)[1]	(✓)[1]	✗[2]	(✓)[1]	✓
programming model	MPI RDMA	MPI	MPI	MPI	PGAS/ GPI	MPI
Reproducibility						
source-code available	✓	✓	✓	✗	✓	✓
still maintained (2024)	✗	✓	✓	✗	✗	✓

Checkpointing libraries either write the program's state to a parallel file system (PFS) or to the compute nodes main memory (*in-memory*; Section 2.8). Checkpointing libraries that save their checkpoints to the PFS include, for example, the algorithm presented by Agarwal *et al.* [Aga+04], FTI [Bau+11], CRAFT [Sha+19], SCR [Moo+10], and VeloC [Nic+19] (Table 4.1). As the number of PEs per parallel program execution continues to grow, the congestion on the PFS increases – resulting in a bottleneck and reduced checkpointing performance [Gos+21; Hér+19]. Thus, some checkpointing libraries have been designed to keep the checkpoints in memory. Examples include ftRMA [BH14], Fenix [Gam+14], `GPI_CP` [Bar+17], and the algorithm described by Lu [Lu05].

All of the above checkpointing libraries employ the substitute strategy (Section 2.7) when recovering after a failure. Therefore, they rely on the availability of replacement nodes, either by requesting additional nodes from the job scheduler, or by keeping spare compute nodes around during fault-free recovery. Keeping spare nodes around, which do not participate in the computation, constitutes a waste of computational resources and energy. Requesting replacement nodes from the job scheduler can incur an overhead of seconds to minutes and is thus impractical. Some checkpointing libraries additionally designate some compute nodes as pure checkpointing nodes, which neither participate at the computation, nor are available as spares to replace failed nodes. As discussed above, we thus advocate for using shrinking recovery, that is, re-distributing the work that was assigned to the failed PEs[1] to the surviving PEs even though this incurs a more involved data recovery mechanism (Section 4.1).

To the best of our knowledge, ReStore is the only in-memory checkpointing library that allows for shrinking recovery. While Ashraf *et al.* [AHE18] describe an implementation of a fault-tolerance mechanism for a specific application that is able to checkpoint to memory and recover in a shrinking setting, this is, however, not a general-purpose checkpointing library

[1] Note that in modern multicore machines, there are multiple PEs per compute node.

but an application-specific approach. Replication approaches similar to the one presented in this chapter are used in distributed fault-tolerant file systems like early versions of the Hadoop Distributed File System [Shv+10] or distributed processing frameworks like Apache Spark [Zah+12]. However, these target very different use cases and sometimes only support rudimentary replication strategies like storing each PE's data on a single partner PE.

4.3.1 Overview of Existing Checkpointing Libraries

In the following, we describe our attempts to replicate the results of competing tools (Table 4.1): The ftRMA [BH14] library has not been maintained since 2014 and relies on the Cray-only foMPI library which has also not been further maintained since 2014. The authors confirmed (pers. comm. 30. June 2022) that the current code exclusively works on Cray systems and is no longer being actively maintained. Although the authors suggested that ftRMA could – in principle – be ported to a non-Cray system, taking into account the unmaintained code base comprising 513 calls to foMPI functions, this would likely incur a prohibitive programming effort with uncertain outcomes. Further, as ULFM currently provides "little support for fault-tolerance" with respect to MPI's Remote Memory Access (RMA) calls [Bou19], deploying ftRMA would be bound to fail using a current, fault-tolerant MPI implementation.

The Fenix library has more than 10 commits in the past year, we thus consider it as still being maintained.[2] However, the author-provided Fenix examples [Gam+14] fail with a segmentation fault on our system. Additionally, its source code and documentation differ substantially among each other. For example, restoring data from another PE is not implemented, and it is unclear which replication scheme is currently employed. As Fenix does currently not support restoring the data that was checkpointed on a different PE, setting up an experimental comparison is challenging. The authors did not respond to our e-mail requesting assistance.

The SCR [Moo+10] repository has more than 50 commits on 20 distinct days during the past six months, and we thus consider it as still being maintained. SCR supports *caching* checkpoints on a RAM-disk. These checkpoints, however, have to be transferred to the parallel file system such as to become available after a PE failure. We therefore do not consider SCR to be an in-memory checkpointing library in the context of node failures.

The source code of Lu's algorithm [Lu05] is not available, and the author can not be contacted, as they did not provide a contact e-mail address on their publications.

The git repository of `GPI_CP` [Bar+17] has only a single commit dating nine years ago; we thus also assert that it is no longer being maintained. Further, as we do not have access to a HPC system where GPI-2 – a library for the Partitioned Global Address Space (PGAS) programming model – is supported, and its dependency `libiverbs` has to be installed by a system administrator, we are unable to compare `GPI_CP` against ReStore.

4.4 In-Memory Replica for Fast Recovery

ReStore allows application developers to store redundant copies of their data in memory, which they can use to create checkpoints or accelerate the redistribution of input data during

[2] This is in contrast to our-original publication of this work [Hüb+22a], where we declared Fenix's maintenance state as "unclear". At that point, the Fenix project had a single commit in six months, and did not pass the author's automated test suite.

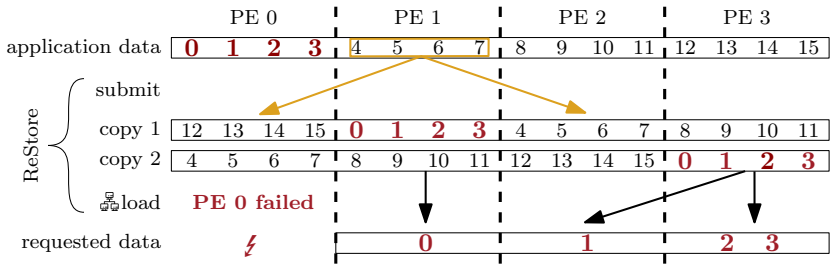


Figure 4.1 Basic data distribution (without block ID permutation) used by ReStore for $p = 4$ PEs, $n = 16$ data blocks, and $r = 2$ copies. The first row shows the data submitted by the application. As an example, the orange arrows show the data ReStore sends from PE 1 to the target PEs, which hold copies of the received data. During a shrinking recovery, the application's load balancer re-distributes the work previously assigned to the failed PE(s) among the surviving PEs, which thus have to load fractions of the respective data. Here, after PE 0 fails, the application requests the data shown in the last row (bold dark red in all occurrences) which is served by ReStore as shown with the black arrows. Note that regardless of the overall PE-count, only r possible senders exist for the data of a failed PE; this constitutes a bottleneck.

recovery (Section 4.1). In case of a failure, the surviving PEs can invoke a recovery routine to load all or parts of the data that was lost during the failure. The application developer can specify which part of the data to load on which PE.

In Section 4.4.1, we introduce our general framework for maintaining redundant copies of the user-supplied data in memory and describe our recovery algorithm. Next, we expand on the distribution of copies by adding random permutations to accelerate the recovery algorithm (Section 4.4.2). Moreover, we analyze the memory usage of our proposed data distribution (Section 4.4.3), and the probability of irrecoverable data loss (Section 4.4.4).

4.4.1 General Framework

The fundamental idea of ReStore is to store r copies of the data on different PEs. By storing them such that it is unlikely for all copies of one data element to fail at once, there will most likely be copies left to recover from (Section 4.4.4 and Section 4.6.2). Users store their data in ReStore using the *submit* function and retrieve data from ReStore using the *load* function. In order to allow a specific PE to request only a fraction of the data, we make the data addressable.

For this, we divide the data into n blocks, each with a unique identifier (ID) x. In its most basic form (Figure 4.1), for $k \in [0, r)$ we store a copy of the block with ID x on PEs

$$L(x,k) = \left\lfloor \frac{xp}{n} \right\rfloor + k \cdot \frac{p}{r} \mod p$$

Under this distribution, by placing the copies of the same data as far apart as possible in the PE-ID (*rank*) space, we expect the different copies of a block to not be stored on the same physical node/case/rack in most cluster setups. This decreases the probability of loosing all copies of a block due to a single failure event, as PEs failures in the same node/case/rack are more likely to occur than a simultaneous failure of unrelated PEs [Bau+11].

During recovery, when PE i requests to load block j – that is not stored in its local part of the ReStore – we choose one of the surviving PEs that holds block j at random to serve the request. If the user requests multiple successive blocks that are all stored on the same set of PEs, we choose one PE to serve all requests. Next, we exchange the requested data using

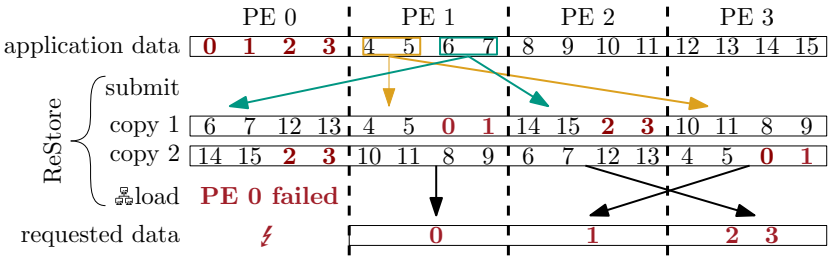


Figure 4.2 Data distribution with block ID permutation used by ReStore for $p = 4$ PEs, $n = 16$ data blocks, $r = 2$ copies, and $s_{pr} = 2$ blocks per permutation range. The first row shows the data submitted by the application. As an example, the orange and green arrows show the data ReStore sends from PE 1 to the target PEs, which hold copies of the received data. During a shrinking recovery, the application's load balancer re-distributes the work previously assigned to the failed PE(s) among the surviving PEs, which thus have to load fractions of the respective data. Here, after PE 0 fails, the application requests the data shown in the last row (dark red in all occurrences) which is served by ReStore as shown with the black arrows.

non-blocking point-to-point messages (Section 2.6). This strategy minimizes the bottleneck number of messages received (Section 2.5). Note that, with r copies for a data block, there are at most r PEs that can act as sources during this message exchange. Further, we copied consecutive blocks, for example $[0, n/p)$, to the same (small) set of PEs. However, we expect that user will commonly assign consecutive block IDs to data blocks stored on the same PE, for example, IDs $[0, n/p)$ on PE 0. Therefore, upon PE failure, the work associated with data blocks with consecutive IDs is redistributed and the respective data blocks are requested. However, all copies of all these blocks (with consecutive IDs) are stored on the *same* (small) set of PE, resulting in a bottleneck. In Section 4.4.2 we explore a modification to this basic distribution scheme, which avoids this bottleneck and thus allows for faster recovery.

Note that using the basic data distribution as described here would locate the first copy of a block on the same PE as we expect the user to commonly assign the respective work package to. For example, we locate the first copy of blocks $[0, n/p)$ on PE 0 – the exact blocks PE 0 will commonly work on. Thus, we shift the distribution of data using an offset. We show this in Figures 4.1 and 4.2, but omit this implementation detail in further discussion.

4.4.2 Breaking Up Access Patterns for Faster Recovery

As discussed above, the data distribution described in Section 4.4.1 induces a bottleneck of only a small set of PE sending data in common use cases. In the following, we explore how to adapt the data distribution such that it avoids this bottleneck, and thus accelerates data recovery, while preserving the failure resilience level.

Assume a failed PE i, which worked on the data blocks $[i \cdot n/p, (i+1) \cdot n/p)$, where n is the total number of data blocks submitted to ReStore. During the subsequent data redistribution, we would ideally desire a dedicated sending PE for each receiving PE, resulting in a bottleneck communication volume of n/p^2 and a bottleneck number of messages sent and received of 1. However, with the data distribution from Section 4.4.1, only the surviving subset of the $r \ll p$ PEs that hold copies of these blocks act as sources, resulting in a bottleneck communication volume of n/pr. We can alleviate this by evenly distributing data block copies $[i \cdot n/p, (i+1) \cdot n/p)$ among multiple PEs, by randomly permuting their block

IDs. For a (pseudo-)random permutation π and $k \in [0, r)$, the block x is stored on PEs

$$L(x,k) = \left\lfloor \frac{\pi(x) \cdot p}{n} \right\rfloor + k \cdot \frac{p}{r} \mod p$$

Thus, if the user requests blocks $[i \cdot n/p, (i+1) \cdot n/p)$, more PEs have parts of the data and can send them to the requesting PEs. This approach, however, can lead to a large bottleneck number of messages being sent and received: If a PE requests n/p^2 data blocks, these blocks can reside on up to $\min(n/p^2, p)$ different PEs. This will require sending many messages of very small size over the network causing a plethora of startup overheads. To mitigate this, we group the data into *permutation ranges* of size s_{pr}. We then apply the random permutation to these permutation ranges instead of applying them to individual data blocks (Figure 4.2).

If a failure of PE i causes the blocks $[i \cdot n/p, (i+1) \cdot n/p)$ to be redistributed, these correspond to the permutation ranges

$$\left[\frac{i \cdot n/p}{s_{pr}}, \frac{(i+1) \cdot n/p}{s_{pr}} \right)$$

If the data are evenly distributed among the $p-1$ surviving PEs, PE j receives the blocks

$$\left[i\frac{n}{p} + j\frac{n}{p \cdot (p-1)}, i\frac{n}{p} + (j+1)\frac{n}{p \cdot (p-1)} \right)$$

Thus, only $n/(p \cdot (p-1) \cdot s_{pr})$ PEs send to every receiving PE.

The best choice of s_{pr}, and whether to use permutations at all, depends on the application's data distribution and which PE requests which part of the data after a failure. It further depends on the frequency of checkpoint creation, as submitting data with permutations enabled results in a more dense communication pattern. We thus experimentally determine a "good" value for s_{pr} (Section 4.6.2).

Note that with this data distribution, we always have sets of $n/(s_{pr} \cdot p)$ permutation ranges that are stored together for all r copies. For example, the blocks $6, 7, 12, 13$ in Figure 4.2 are always stored together – once on PE 0 and once on PE 2. This means that for all copies of any permutation range whose first copy is stored on PE i to become lost, exactly the set of r PEs $i + k \cdot \frac{p}{r}, k \in [0, r)$ has to fail. One could also opt for a different approach – for example, using a distinct permutation for each copy. In this case, no sets of permutation ranges will always be stored together. So, in order for any permutation range whose first copy resides on PE i to be lost, it is sufficient if *any* of the $n/s_{pr}p$ sets of PEs of size r fail that hold the copies of one of the permutation ranges. Note that this yields a higher probability for irrecoverable data loss than our proposed distribution. In Section 4.4.4, we analyze the probability of irrecoverable data loss under our proposed data distribution.

4.4.3 Memory Usage

Other fault-tolerance libraries [Bau+11; BH14; Lu05] reduce their memory footprint via erasure coding – for example the Reed-Solomon code [RS60]. That is, they do not store the replicas A' and B' of two blocks A and B but rather the XOR of these blocks $A \oplus B$, thus halving the required storage for the checkpoint. However, erasure coding incurs additional messages upon checkpoint creation and recovery as well as a substantial computational overhead [CD96]. Therefore, we do not encode checkpoints in ReStore with an erasure coding scheme, thus trading reduced communication overhead for increased memory consumption.

Again, let n be the number of data blocks, r be the number of replicas, and p be the number of PEs. On each PE, ReStore requires main-memory to store $r \cdot n/p$ data blocks for the replicated storage. The memory requirement is doubled during submission as we require additional space for the send and receive buffers. During recovery, an additional copy of all data being sent and received is stored on each PE. We verified these formulas empirically (data not shown). A plethora of applications exist for which the amount of memory for the input data *and* the data that need to be checkpointed fit in memory r times. Examples include RAxML-NG [Hüb+21b; Koz+19], k-means, and PageRank.[3] For example, RAxML-NG is memory bandwidth bound [Koz18]. Hence, using additional cores, with their associated larger cache memory capacity, can even yield super-linear speedups due to increased cache-efficiency. Such applications can therefore substantially benefit from the reduced communication and computational overhead we achieve by creating and restoring a checkpoint without erasure codes.

4.4.4 Probability of Irrecoverable Data Loss

Let r be the replication level (number of copies) and p be the number of PEs. In this analysis, we assume that $r|p$ (r divides p), which constitutes a reasonable assumption for $r = 4$ and current two socket systems, which comprise an even number of cores per socket. If $r|p$, the PEs are divided into $g = p/r$ groups, with all PEs in a respective group storing the same data. Thus, if and only if, all r PEs in a specific group fail, we will not be able to recover a part of the data. We denote such an event as *Irrecoverable Data Loss (IDL)*.

Let f be the number of failed PEs. There is exactly one possibility to draw r out of r PEs belonging to a single group. The number of possibilities to draw the remaining $f - r$ failed PEs among the remaining PEs such that they do *not* belong to the given group is

$$C_{f-r}^{p-r} := \binom{p-r}{f-r}$$

The overall number of possibilities to draw f PEs from the p PEs that are still alive at program start is C_f^p. Thus, the probability that, provided f failures, all processes of a *given* group fail is $1 \cdot C_{f-r}^{p-r}/C_f^p$. When generalizing this equation to the probability of all PEs of *at least one* group failing, we have to apply the inclusion-exclusion principle in order to avoid counting the same combination multiple times. We thus obtain the following equation for the probability of an IDL occurring at failure f or any failure before:

$$P_{\text{IDL}}^{\leq}(f) = \sum_{j=1}^{g} (-1)^{j+1} C_j^g \frac{C_{f-jr}^{p-jr}}{C_f^p}$$

Therefore, the probability of an IDL occurring at exactly failure f is given by

$$P_{\text{IDL}}^{=}(f) = P_{\text{IDL}}^{\leq}(f) - P_{\text{IDL}}^{\leq}(f-1)$$

Thus, the expected number of failures until an IDL occurs is

$$E[\text{Failures until IDL}] = \sum_{f=r}^{p} P_{\text{IDL}}^{=}(f) \cdot f$$

[3] We implemented fault-tolerant version for all three of these using ReStore and show running times for RAxML-NG and k-means in Section 4.6.3.

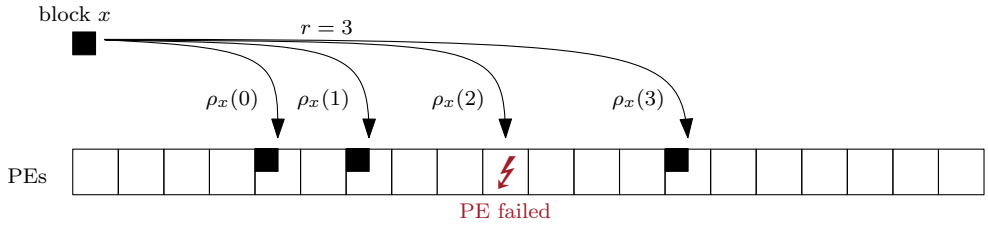


Figure 4.3 Modified data distribution enabling easy recovery of replicas after a node-failure. r denotes the number of replicas. Each block x is placed on the first r PEs in the permutation of all ranks ρ_x. Note that ρ_x depends on the block ID x, thereby spreading out the blocks that resided on a single failed PE across multiple surviving PEs.

Further, for small f, we can approximate the probability of an IDL by $P^{\approx}_{\text{IDL}}(f) = g \cdot (f/p)^r$ Solving for the fraction of PEs that fail f/p such that $P^{\approx}_{\text{IDL}} = 1$, yields $f/p = (r/p)^{(1/r)} \in \mathcal{O}(p^{-1/r})$ for a fixed r. In order to empirically verify these formulas, we simulate PE failures using the actual data distribution (Section 4.6.2).

4.4.5 Recovering Lost Replicas After a Node Failure

To further increase the resilience of our framework, we introduce an approach that allows to restore replicas that were lost during a PE failure while keeping all other replicas in place. That is, we do not need to redistribute any replicas that reside on surviving nodes, thus reducing the required amount of data to be exchanged to the theoretical minimum. As in the previous sections, let n be the number of data blocks, r be the number of replicas per block, f be the number of failed PEs, and p be the number of overall PEs.

We design a data distribution scheme that requires only $\mathcal{O}(1)$ extra space – thereby ruling out lookup-tables – and is able to output the list of PEs a data block x is stored on in $\mathcal{O}(r + f)$ time. For this, we draw a different random permutation ρ_x of $[0, p-1]$ (or long, non-repeating random sequences of PEs) for each block x and place the replicas of x on the first r alive nodes of that permutation (Figure 4.3). When a PE fails, we copy all replicas that this PE held to the next PE in each replica's permutation. This ensures that the blocks are (stochastically) evenly distributed across the surviving PEs. We can refine this approach to attain a perfectly balanced initial data distribution and reduce the probability of an IDL (Section 4.4.2). For this, we initially place the first r replicas $(L(x,0), L(x,1), \ldots, L(x,r-1))$ deterministically as described in Section 4.4.1. That is, the data distribution is given by

$$L(x,k) = \begin{cases} \text{as described in Section 4.4.1,} & \text{if } k < r \\ \rho_x(k), & \text{else} \end{cases}$$

In order to maintain a fast recovery, we apply this technique on a permutation-range-level rather than on individual blocks as explained in Section 4.4.2. Below, we describe three different options to draw the required random sequence of PEs for every block x.

Double Hashing We can generate different permutations for each block x, via a scheme that is analogous to collision resolution in hash tables using open addressing and double hashing [San+19, ch. 4.3]. Let $\bar{h}$ and h_s be collision-avoiding hash functions, and s be a seed to parameterize h_s. For $k \in \mathbb{N}_0$, we define the permutation of PEs for a block x as

$$\rho_x(k) = (\bar{h}(x) + k \cdot h_s(x)) \mod p$$

Concretely, in order to obtain the list of PEs, where the r copies of block x are currently stored on, we evaluate $\rho_x(k)$ starting with $k = 0$ until we find r non-failed PEs. The probing sequences ρ_x and ρ_y are likely to be different for two blocks x and y, even if $\rho_x(0) = \rho_y(0)$.

If $L(x, k) = L(x, j)$ for $k < j < p$, the entire sequence will repeat from that point on, yielding recovery of lost replicas impossible after more than j failures. Thus, in order to avoid this, we require the seeded hash function h_s to only yield integers that are coprime to p. This can be implemented by examining different seeds s for h_s until $h_s(x)$ is coprime to p. For this purpose, we draw a sequence of seeds uniformly at random at program startup. The probability of two random integers being coprime is $6/\pi^2$ [HW60]. Thus, the expected number of differently seeded hash functions we need to evaluate for a random[4] p is

$$1 + \sum_{n=1}^{\infty} \left(1 - \frac{6}{\pi^2}\right)^n = \frac{7}{6}\left(\pi^2 - 6\right) \approx 1.65$$

To check if $h_s(x)$ is coprime to p, we divide $h_s(x)$ by every prime factor of p, which we factorize once during program startup. The Erdős-Kac theorem [EK40] states that the number of distinct prime factors m of a random number below $\hat{p}$ approximately follows the normal distribution with a mean and variance of $\log\log\hat{p}$. For example, a node count of $p \approx 10^9$, has $m = 3 \pm 1.7$ expected prime factors. For node counts $p < 10^9$ ($m = 3$) we therefore expect less than $m \cdot 1.65 = 5$ divisions to check for coprimality each time we need to compute on which nodes to store a block.

Feistel Networks Alternatively, we can deploy a classical seeded pseudorandom permutation ρ_s on $[0, p-1]$ to generate an independent probing sequence for each element x. Note that in particular, encryption systems are permutations. For example, we can employ a Feistel Network as our permutation ρ_x. We seed the Feistel network $\mathcal{F}$ for each block x with $\bar{h}(x)$, where $\bar{h}$ again, is a collision avoiding hash-function and define

$$\rho_x(k) = \mathcal{F}_{\bar{h}(x)}(k)$$

Linear Congruential Generator A Linear Congruential Generators (LCGs) is a function which yields a pseudorandom sequence of integers in a configurable range $[0, m)$:

$$LCG_{a,c}(k+1) = (a \cdot LCG(k) + c) \mod m$$

LCGs are known to generate each number in their value range exactly once before repeating, if their parameters a and c are chosen according to the Hull-Dobel theorem [HD62].[5] Given a hash function $\bar{h}$, we can define a permutation seeded with the block ID x:

$$\rho_x(k) = \left\lfloor LCG_{a,c}(k + \bar{h}(x)) \cdot \frac{p-1}{m-1} \right\rfloor$$

4.5 Implementation

We implement ReStore as a C++ library[6] using the ULFM proposal implementation [Bla+13]. Application programmers submit their data to ReStore by writing their serialized data blocks

[4] Though realistically, we cannot expect p to be random.
[5] Which is trivial, if m is a power of two.
[6] `https://github.com/ReStoreCpp/ReStore`; LGPL

to a memory location supplied by the library or using ReStore's interface for already serialized data. After a failure, applications can request data blocks by passing a list of block identifier ranges to ReStore. This can be done in two ways: Either by providing the full list of requested block IDs to all PEs or by providing exactly those ID ranges each individual PE requires on exactly that PE. By using the first approach, no communication is required to determine which PE serves which request. When using the second approach, the receiving PE will determine which PE should send each requested data block. Subsequently, the PEs exchange the respective data in what is conceptually a sparse all-to-all (implemented as non-blocking point-to-point messages). Preliminary experiments showed that the latter method performs substantially better because the full list of requests usually scales with the number of PEs in the application, slowing down the first approach. Thus, for all experiments in Section 4.6, we employ the second approach.

4.6 Experimental Evaluation

In the following, we first tune ReStore's parameters. For this, we empirically determine the ideal number of bytes per permutation range w.r.t. runtime performance and verify that permuting the block IDs incurs a speedup over using consecutive IDs (Section 4.4.2). Further, we decide on a replication level to use in our experiments based on the probability of IDL occurring depending on the number of replica and failures. Next, we demonstrate ReStore'a real-world applicability by integrating it into two fault-tolerant applications: A simple k-means algorithm and a complex bioinformatics application, RAxML-NG, used by thousands of researchers (Section 4.6.3). Finally, in Section 4.6.4 we compare ReStore to reading from a PFS – which represents a lower bound for checkpointing libraries using the PFS as storage – as well as the reported running times by other checkpointing libraries.

4.6.1 Environment and Experimental Setup

We run our experiments on the SuperMUC-NG. Each node consists of two Intel Skylake Xeon Platinum 8174 processors with 48 cores and 96 GiB of memory each. The compute nodes communicate via an OmniPath network with a bandwidth of 100 Gbit s^{-1}. The operating system is SUSE Linux Enterprise Server 15 SP1 running Linux Kernel version 4.12.14-197.78. We compile our benchmark applications using GCC version 10.2.0 with full optimizations enabled (`-O3`) and all assertions disabled. Unless otherwise stated, we communicate using OpenMPI version 4.0.4. We verify that our implementation *does work* if nodes actually fail and communication is recovered with ULFM as part of our fully automated unit tests. At the time, the current version of ULFM, however, was not sufficiently stable to conduct reliable performance benchmark experiments.[7] For example, processes were sometimes reported incorrectly as failed or recovery split the processors into two separate groups, each assuming the other group's failure. We reported this behavior on the ULFM mailing list and the authors of ULFM reproduced and confirmed the bug.[8] Additionally, most communication and fault tolerance mechanisms were slow (see Hübner *et al.* [Hüb+21b] for details). In our performance benchmarks, we thus use OpenMPI and simulate failures by removing processes from the calculation using `MPI_Comm_split` and replacing other required fault recovery steps by functionally similar ones (e.g., replacing `MPIX_Comm_agree` with `MPI_Barrier`s). All plots

[7] ULFM has now been merged into OpenMPI 5.0.
[8] George Bosilca. Post `pbSToy94RhI/xUrFBx_1DAAJ` on the ULFM mailing list.

show results for 10 repetitions per experiment. Plots depict the mean with error bars for the 10th and 90th percentile.

4.6.2 Isolated Evaluation

In this section, we evaluate ReStore in an isolated environment. First, in Section 4.4.1, we empirically determine the optimal number of redundant copies to create for each data block, a tradeoff between memory usage (Section 4.4.3), checkpoint creation, and restoration runtime. Next, we analyze ReStore's runtime performance and experimentally determine the ideal size of permutation ranges (Section 4.4.2).

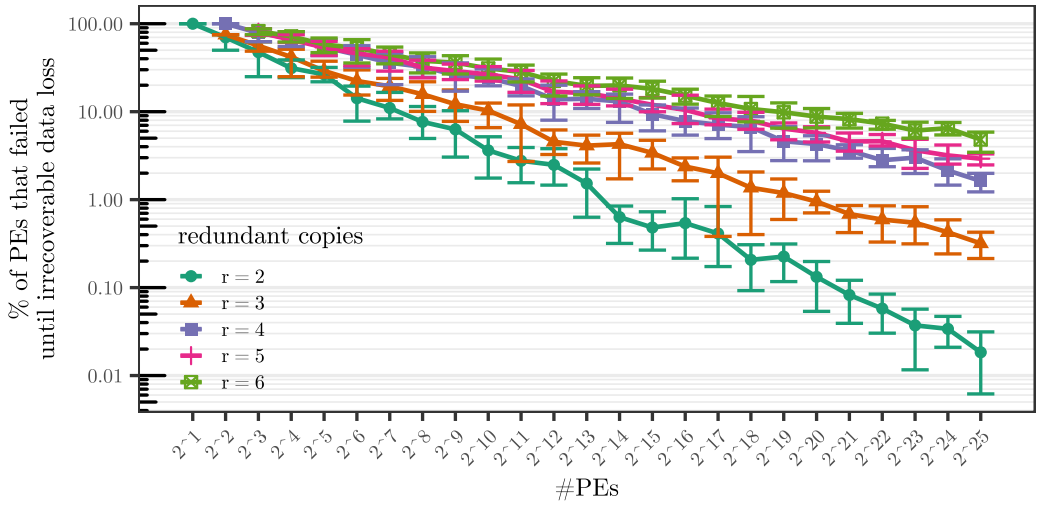


Figure 4.4 Number of PE-failures until at least one data block lost all its redundant copies. We simulate the failure of random PEs until there is at least one data block with no remaining copies on the surviving PEs. We then plot the fraction of PEs that failed. We rerun the simulation 10 times, showing the median, 10th, and 90th percentile.

Number of Redundant Copies We empirically determine the number of redundant copies r one needs to keep for each data block. For this, we simulate PE-failures using ReStore's actual data distribution (Section 4.4.2) until at least one block loses all of its redundant copies because they resided on failed PEs (Figure 4.4). We report, that even for 2^{25} PEs, more than 1 % of all PEs have to fail until we can no longer recover all data when using $r = 4$ redundant copies. For applications running on fewer PEs, an even smaller number r of redundant copies is sufficient to yield data loss unlikely. In the event of an irrecoverable data loss, the program has to re-load its static input data from disk or restart its computation from scratch – depending on weather ReStore was used to accelerate the redistribution of static data or to checkpoint dynamic data. For all further experiments we therefore set the number of redundant copies to $r := 4$. Further, we compare the values obtained by applying the formulas described in Section 4.4.4 with the values obtained via simulation (Figure 4.5) showing that our theoretical formulae match the simulation results very closely.

Block ID Randomization In the following, we first empirically optimize the size of our permutation ranges (Section 4.4.2) and subsequently evaluate the runtime of submitting and

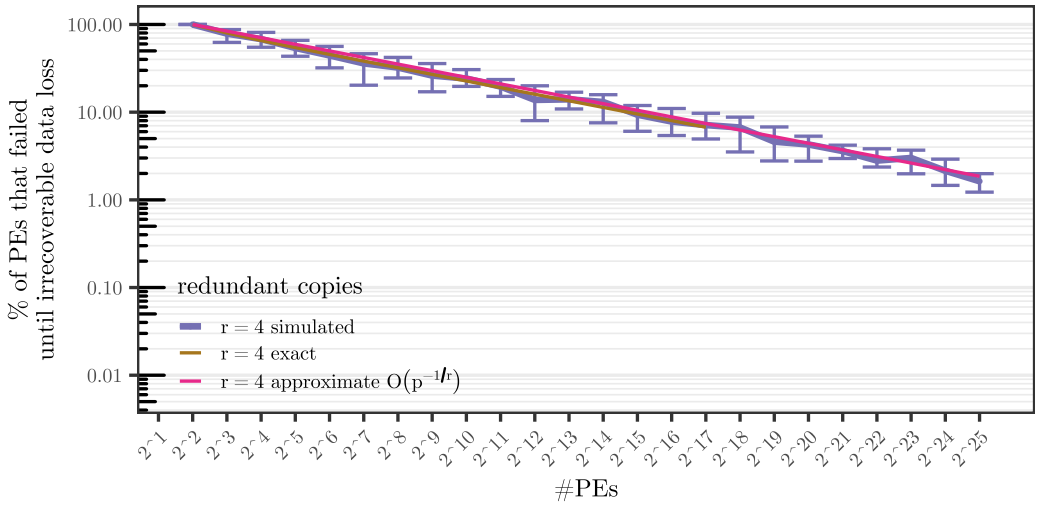


Figure 4.5 Number of PE-failures until at least one data block lost all its redundant copies. We compare the theoretical probability (Section 4.4.4) and the simulated values from Figure 4.4.

restoring data with ID permutations turned on and off, respectively. For these experiments, we use data blocks of size 64 bit (e.g., one `double`) and 16 MiB of data per PE. We show results for three different operations: In the `submit` operation we pass 16 MiB of data on all PEs to ReStore's submit function. In `load 1 % of data`, we load the data submitted by 1 % of the PEs with contiguous data block IDs evenly across all PEs. So, for n total data blocks, we pick a random starting PE i and request data blocks $i \cdot n/p$ to $(i + 0.01 \cdot p) \cdot n/p$. This simulates the requests expected if 1 % of PEs fail at once. In `load all data` we load all data stored in ReStore evenly distributed across all PEs in a way that no PE loads the same data it originally submitted. This constitutes the worst case running time scenario.

By decreasing the size of permutation ranges (Section 4.4.2) we can control how many PEs can participate in sending requested data: When using smaller permutation ranges, more PEs are able to serve parts of the requested data to the requesting PEs. Small permutation ranges, on the other hand, lead to fragmentation of the data and therefore require numerous small messages to be exchanged over the network. In Figure 4.6, we show the number of bytes per permutation range on the x-axis and the running times of `submit` and `load 1 % of data` on the y-axis for different numbers of PEs. We do not show results for `load all data` because permutations even have a negative effect on performance here (Figure 4.7). This is, because all PEs can participate in sending the data even without permutations. Therefore, they only induce additional messages (see below). We hence recommend turning them off when using a recovery mechanism which loads all data stored in ReStore. We observe that for few bytes per permutation range, both `submit` and `load 1 % of data` are slower by up to an order of magnitude than the fastest configuration because of a high bottleneck number of messages. Approaching 16 MiB of data per permutation range, fewer PEs can participate in sending data. Between these two extremes, there is a range of permutation range sizes which yield fast running times. For all subsequent experiments, we thus fix the amount of data per permutation range to 256 KiB (0.65 to 2.27 ms for `load 1 % of data` on 48 to 6144 PEs). For 16 MiB of data per PE, every PE receives approximately 164 KiB in `load 1 % of data` which results in an average of two PEs requesting the same permutation range and therefore induces a sparse communication pattern. On the sending side, this implies that

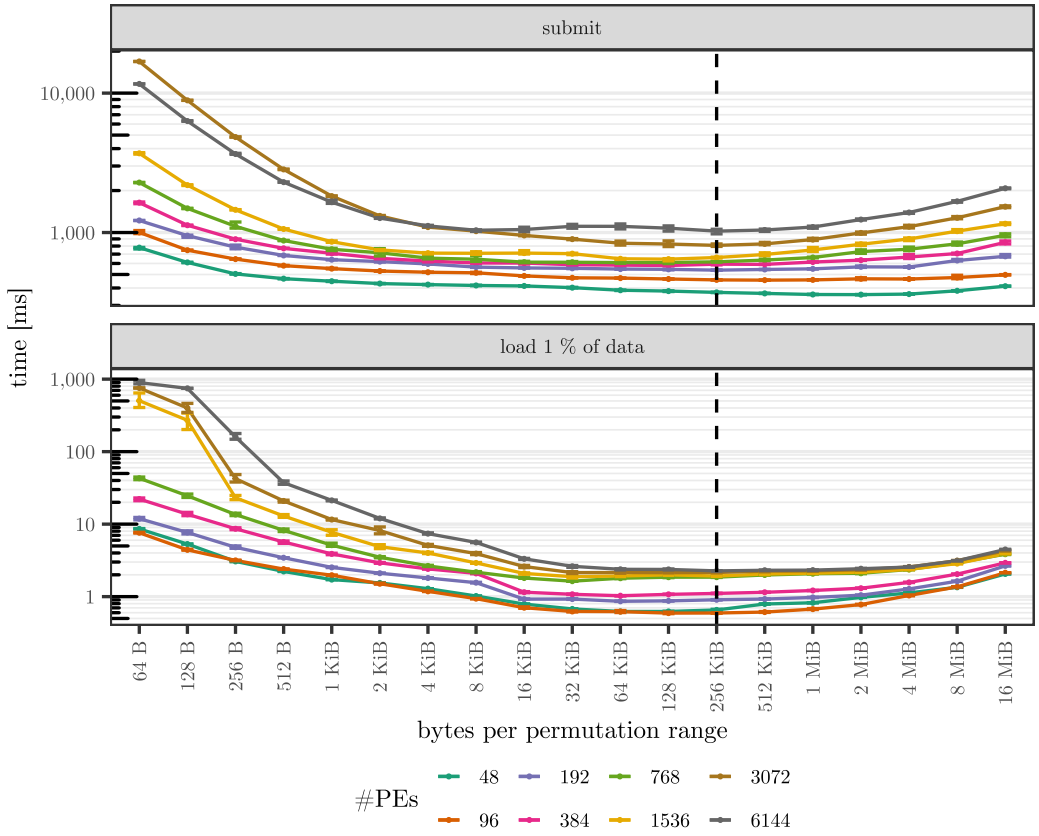


Figure 4.6 Optimizing the number of bytes per permutation range. The runtime of submitting data to and loading data from ReStore depends on the number of bytes which are being permuted together as described in Section 4.4.2. Fewer bytes permuted together will spread out the data more, yielding more, but shorter messages. We chose 256 KiB (dashed line) as the default value.

the data submitted by a single PE is distributed among 64 permutation ranges. With $r = 4$ redundant copies, this results in up to $64 \cdot 4 = 256$ PEs that participate in serving this part of the data.

As expected, enabling random permutations speeds up `load 1 % of data` and slows down `load all data`, especially for runs on many PEs (Figure 4.7). This is because in `load all data`, even without permutations, every PE needs to send some part of the data. By enabling permutations, the data requested by a PE is distributed among more sending PEs, resulting in a denser communication pattern. We can tolerate an increase in running time of `submit` as it is invoked only once in the case of only submitting input data. In contrast, a load is issued after every failure. We therefore provide the application developer with a tuning parameter to adapt the behavior of ReStore to their application's access pattern.

4.6.3 Applications

To demonstrate realistic ReStore use cases, we employ it to restore lost data in two different real-world applications. Figure 4.8 shows running times for a small example application that

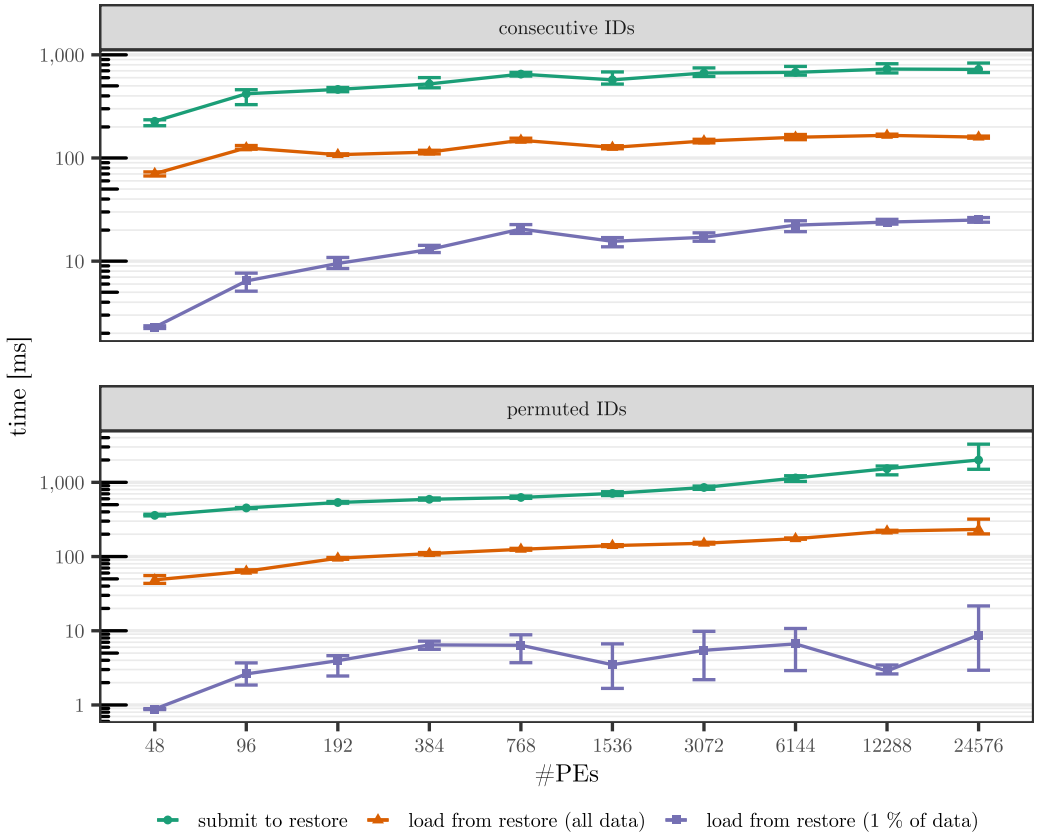


Figure 4.7 Weak scaling experiment (16 MiB per PE) of our three benchmark operations with and without ID randomization. We copy all data over the network – that is, no PE holds a copy of its requested data in its local part of the ReStore storage.

computes a k-means clustering [Mac67].[9] Each PE holds 65 536 points in a 32-dimensional space as input with 64 bit double precision floating-point values per dimension resulting in 16 MiB of input data. All PEs start with the same 20 random starting centers. Iteratively, each PE assigns the nearest center to each of its local points and all PEs collaboratively calculate new center positions using an all-reduce operation (Section 2.6) over k elements. If a PE fails, the remaining PEs divide the failed PE's data points evenly among themselves using ReStore and continue with the calculation. We perform 500 iterations of the algorithm and simulate an expected failure of 1 % of all nodes distributed uniformly at random during these iterations. This is done by determining a suitable probability for each PE to fail in each iteration of the algorithm – implemented via a discrete exponential decay with 1 % of failed PEs after 500 iterations. We find that ReStore accounts for only 1.6 % (median) of the overall running time on up to 24 576 PEs with up to 262 PEs failing. Note that the overall running time increases by more than ReStore's overhead for large PE-counts. This is mainly due to MPI operations required to restore a functioning communicator after a PE failure.

[9] We ran these experiments with Intel MPI, because its `Group_*`-functions – which we use to determine which PEs failed – perform better than OpenMPI's.

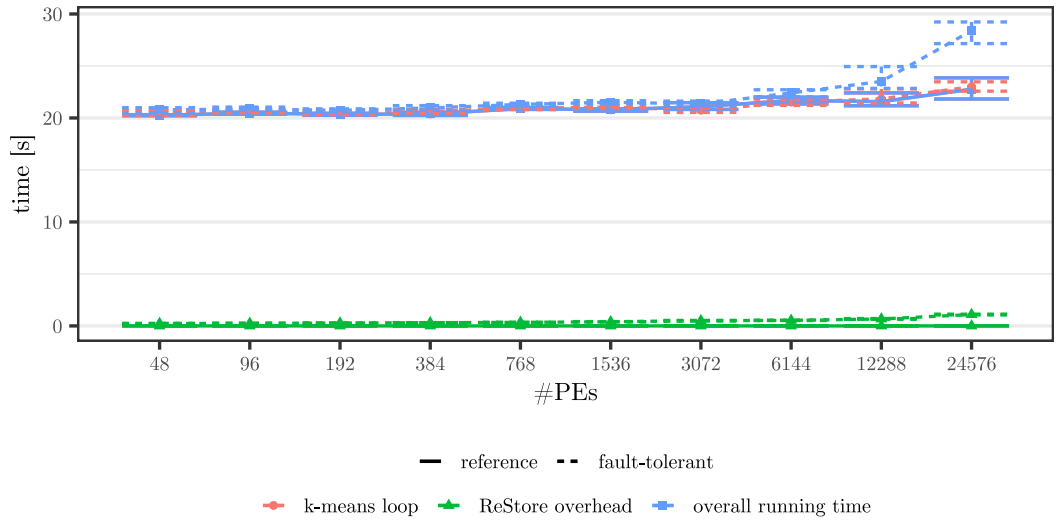


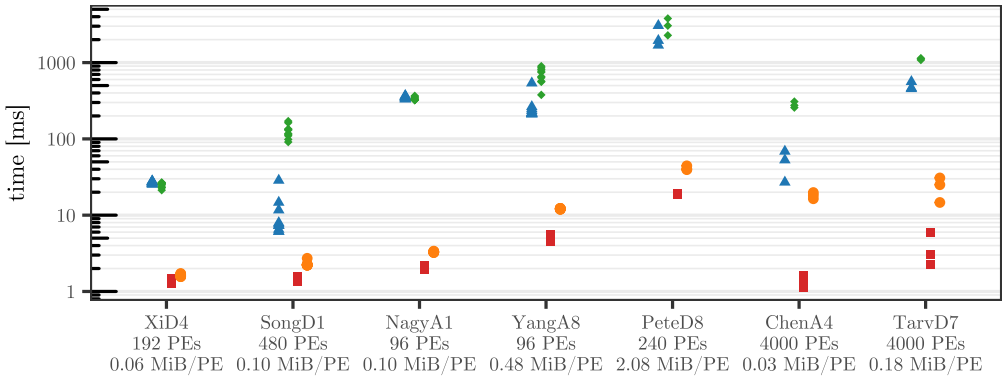
Figure 4.8 Running time of the k-means clustering algorithm with and without failure of 1 % of PEs on 16 MiB of data per PE. *k-means loop:* time spent for the core clustering algorithm. *ReStore overhead:* time spent in ReStore's functions. *Overall running time* also includes additional work required for attaining fault-tolerance, such as a load balancer to determine how to redistribute data and MPI functions for identifying the failed PEs.

Next, we demonstrate ReStore's performance for the fault-tolerant version of the highly complex and widely-used phylogenetic tree inference tool RAxML-NG [Koz+19] – called FT-RAxML-NG [Hüb+21b] – using the same empirical datasets as in [Hüb+21b] (Figure 4.9a). Additionally, we use a 19.1 GiB synthetic dataset [AKS14] for scaling experiments (Figure 4.9b). FT-RAxML-NG redistributes its input data among all surviving PEs. We therefore deactivate permutation ranges for this application. We compare ReStore's performance against FT-RAxML-NG's currently implemented recovery mechanism: Loading the data from the PFS using RAxML-NG's dedicated binary file format (RBA), which enables rapidly reading only the required subset of the input matrix. We distinguish between the input files being uncached by the file system (in the first read) and being cached by previous reads. Both, submitting data to ReStore and loading data after a failure, is faster than the original method of loading the data from files – often by more than one order of magnitude. On the synthetic dataset, for low PE-counts, submitting to ReStore is slower than reloading from a file. However, this is negligible because an actual phylogenetic inference on this dataset requires terabytes of memory for likelihood calculations and would therefore never run on this few PEs. We also emphasize that submitting to ReStore has to be done only once in FT-RAxML-NG, while loading has to be conducted after every failure.

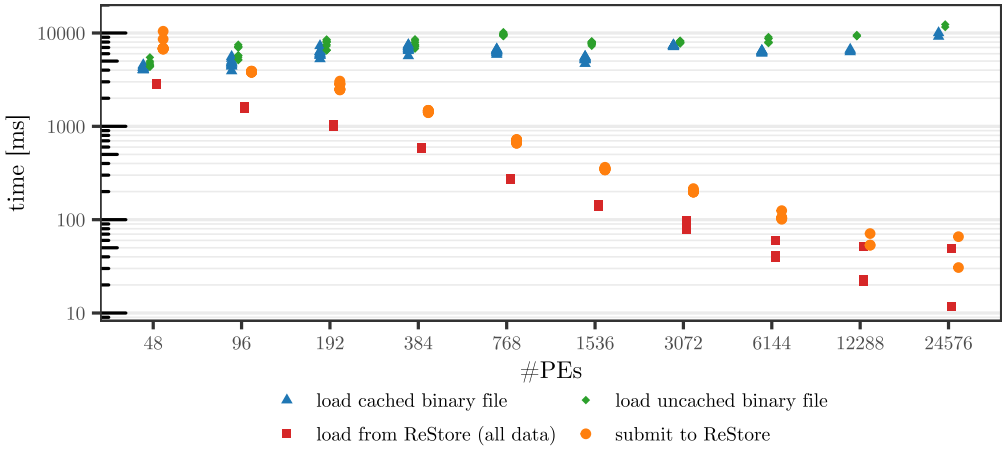
Further, as the probability of PE-failure increases with the number of PEs, we require checkpointing libraries whose running time not only remains constant, but decreases with additional PEs. For the weak scaling experiment on the synthetic dataset, the time required for reading data from the PFS *increases* with the number of PE, while the time required to submit data to, and load data from, the ReStore *decreases* substantially (Figure 4.9b).

4.6.4 Comparison with Other Approaches

Next, we evaluate ReStore's performance in comparison to other checkpointing approaches. However, as outline in Section 4.3, most checkpointing libraries store their checkpoints on



(a) Real world datasets



(b) 19.1 GiB synthetic dataset

Figure 4.9 Running times to re-load data after the necessary re-distribution of work following a shrinking failure-recovery in FT-RAxML-NG. RAxML-NG either loads the data from ReStore, or from a binary file located on the parallel file system, which either can be in the file server's cache or not. **(a)** Labels on the x-axis show the name of the dataset, the number of PEs used for the respective dataset, and the corresponding amount of input data per PE.

the PFS. Further, most on-disk and all in-memory checkpointing libraries only support substituting recovery, that is, no shrinking recovery (Section 2.7), yielding a comparison with ReStore challenging. Additionally, as detailed in Section 4.3.1, to the best of our knowledge, there exists no in-memory checkpointing library that is still being maintained and working to compare ReStore to.

Comparing to Disk-Based Approaches We compare ReStore against loading a copy of the data stored on the PFS (Figure 4.10). We create the respective file such that we can read all data in a single consecutive read operation and therefore as fast as possible. We show running times for reading a separate file for each reading PE using C++'s `ifstream` and reading a single file for all PEs via `MPI_File_read_at_all` (MPI I/O in the plot). This constitutes a lower bound for all checkpointing libraries that need to read their data from disk. ReStore outperforms disk access (`ifstream`) on 24 576 PEs by a factor of 206 (1 % of application data; median) and 55 (all application data; median), respectively. For

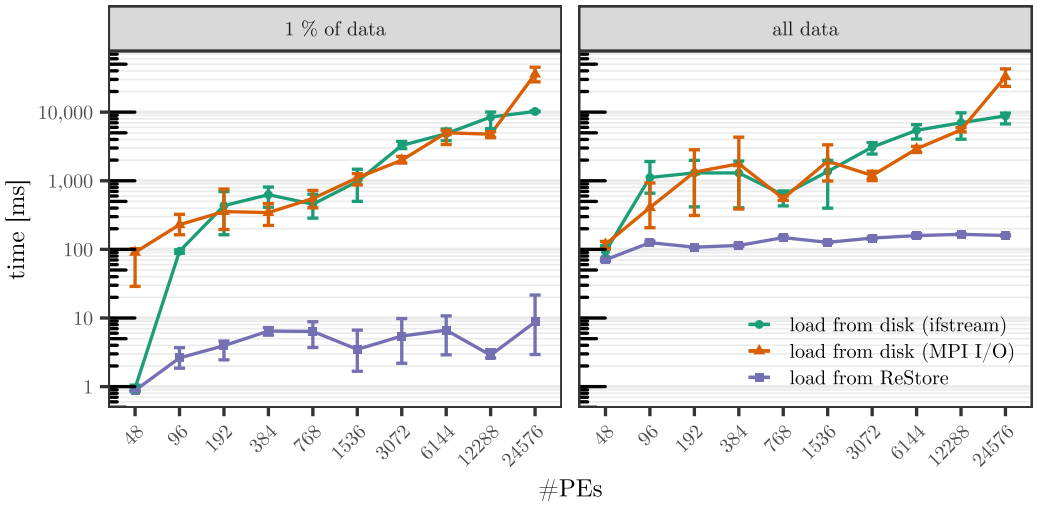


Figure 4.10 Loading performance of ReStore vs. loading from files on the HPC system's parallel file system, representing the approach of most checkpointing libraries.

48 PEs (1 compute node of the HPC system), reading from file seems to be almost as fast as ReStore. This might be explained by the file system caching the file on the same node as we write it immediately before reading it – usually on a different node which is not possible using only 1 node. Note, however, that all PEs on a single node are likely to fail at once, rendering in-memory checkpointing futile on a single node. We conclude that in-memory checkpointing as implemented in ReStore will substantially outperform any disk-based approaches, especially when reloading only fractions of the data, such as during shrinking recovery.

Comparing to Reported Measurements As we were unable to obtain working competing in-memory checkpointing libraries, despite substantial (four person-weeks) effort (Section 4.3.1), we compare ReStore to the reported measurements of the respective papers in the following.

Besta and Hoefler [BH14] benchmark ftRMA by measuring the slowdown it induces on two existing programs, but not in isolation and are thus challenging to compare to. Gamell *et al.* [Gam+14] measure approximately 115 ms to write a checkpoint with 14.8 MB per PE on 1000 PEs using Fenix. Fenix implements a replication level of $r = 1$. This means, that there exists a single copy of the data in addition to the data the PEs are actively working on. According to our definition (Section 4.4.4), a single PE failure will cause irrecoverable data loss. This works in practice, as long as the data which resided on the failed PE(s) does not need to be restored. In comparison, to serialize and store 16 MiB per PE on 1536 PEs (32 nodes) with a replication level of $r = 1$ and using consecutive IDs, ReStore requires 126 ± 3 ms ($\mu \pm \sigma$, 10 repeats). Gamell *et al.* [Gam+14] expect Fenix's recovery time to be analogous to its checkpointing time but do not provide experimental results for this claim. In comparison, ReStore restores the data of a single PE to another single PE in our experiments in 21 ± 2 ms. ReStore additionally offers restoring data of a single PE scattered to all surviving PEs. This operation requires 20 ± 5 ms in our experiments. Further, in the case that one expects more than one recovery per checkpoint, ReStore offers ID permutations to speed up recovery at the cost of slower checkpoint creation (Section 4.4.2). This would, for example, be the case for static input data, of which multiple PEs need different, but

small fractions of data after getting assigned new work following a PE failure. With ID permutations enabled, saving the data to ReStore requires 215 ± 9 ms in our experiments. Restoring the data requires 15 ± 3 ms if restoring all data to a single PE and 0.9 ± 0.2 ms if restoring the data scattered across the surviving PEs. As the latter evenly distributes the data across the surviving PEs, we expect it to become the more common scenario when working in a shrinking setting. Further, Bartsch *et al.* [Bar+17] report `GPI_CP` to require approximately 1 s to initialize, 200 ms to create a checkpoint, and 15 ms to restore data from a checkpoint.

Fenix's performance was measured on a Cray XK7 system with 16 cores per node and a 160 GB s^{-1} network [Cra13]. `GPI_CP`'s performance was measured on an unnamed system with 16 cores per node and a QDR Infiniband network. We measured ReStore's performance on the SuperMUC-NG, which has 48 cores per node and an OmniPath interconnection with 100 Gbit s^{-1} (Section 4.6.1). We choose our experiments such that data are always copied between different nodes and never between two processes running on the same node. Thus, all 48 processes on a single node have to share the same interconnect. Considering that we evaluate ReStore on a slower network than Fenix, we expect an even more favorable comparison when having access to a similar HPC system.

Lu [Lu05] reports checkpoint creation times of 8 to 20 s for 157 to 182 MB on 448 PEs. They report restoration times of 20 to 48 s. Thus, assuming linear scaling, we expect checkpoint creation times of approximately 1 s and restoration times of approximately 2 s for 16 MiB of data. Lu's algorithm is thus an order of magnitude slower than ReStore and Fenix. We assume this is due to the fact, that Lu's algorithm uses erasure codes (Section 4.4.3).

To summarize, ReStore can be configured to create and restore from checkpoints in the same manner and approximately the same time as existing checkpointing solutions. ReStore additionally has functionality to (a) increase the replication level (b) restore the data in a scattered manner to multiple PEs instead of to only one PE, and (c) enable ID permutations to decrease data restoration time by an order of magnitude while doubling the time required to create a checkpoint. The latter option is useful, for example, when creating a replicated storage for the input data of a program, which has to be partially reload after a failure.

4.7 Conclusion and Future Work

Mitigating hardware failures constitutes a major challenge in High Performance Computing [SDM10]. As it is impractical to either, request replacement for failed PEs or to maintain spare PEs, applications must continue execution with the remaining ones (shrinking recovery). Thus, during failure-recovery, applications have to re-distribute their workload and reload the respective parts of the input data to the remaining PEs. However, current checkpointing libraries often do not support this data redistribution. Further, they typically write checkpoints to the parallel file system instead of the main memory, inducing slow recoveries due to low disk access speeds and inducing disk congestion.

We present ReStore, the to the best of our knowledge, first algorithmic framework that enables shrinking data-recovery from memory when mitigating hardware failures in MPI programs. This allows applications to employ all PEs for the computation, instead of having to maintain some of them as spares for replacing failed PEs.

We evaluate ReStore in both, controlled, isolated environments, and using real-world applications. Our experiments show loading times of lost data in the range of milliseconds on up to 24 576 PEs. We achieve this by using a distribution scheme for redundant copies that ensures a low probability of data loss and allows for rapid data recovery by balancing the

bottleneck communication volume and bottleneck number of messages. Further, we analyze the probability of irrecoverable data loss and propose a data distribution to easily restore lost replicas after a failure. ReStore can be used by any HPC applications using MPI 5.0 – which introduces fault-tolerance mitigation functionality. For example, using ReStore, we were able to improve recovery performance of FT-RAxML-NG [Hüb+21b; Koz+19] by up to two orders of magnitude. Therefore, ReStore enables scientific applications to mitigate hardware failures in milliseconds, without requiring replacement or spare resources.

In future work, we plan to provide an experimental and theoretical evaluation of the proposed recovery of lost replicates after a PE-failure (Section 4.4.5). This will further decrease the probability to lose all copies of any datum. Further, next to its usage in fault-tolerant algorithms, we also plan on evaluating if ReStore can facilitate rapid re-distribution of work in the context of load-balancing in distributed applications.

5 Bit-Reproducible Reduction Under Varying Core-Counts in Distributed Phylogenetic Inference

Attribution: This chapter is based on work conducted in close collaboration with Christoph Stelz, who explored this topic during his Bachelor's Thesis, which I supervised. Subsequent to his thesis work, we continued to work on this project in close collaboration. Our enhanced understanding of the factors influencing runtime gained from his thesis, including profiling insights, led me to design a novel bit-reproducible reduction algorithm, which Christoph Stelz implemented and integrated into our bit-reproducible RAxML-NG variant. Additionally, I designed, conducted, and assessed the reproducibility study for phylogenetic tree inferences. Moreover, I designed the runtime benchmarks for ReproRed and Repro-RAxML-NG in close collaboration with Christoph Stelz, who wrote and executed the benchmarks. I then analyzed the results and created the visualizations. I authored the *text* in this chapter, with Christoph Stelz providing feedback and filling in details. This work is currently unpublished.

Summary: Phylogenetic trees describe the shared evolutionary history among related biological species based on their genomic data. Maximum-Likelihood based phylogenetic tree inference tools search for the tree and evolutionary model that best explain the observed genomic data. Given the independence of the likelihood score calculations at different genomic loci/sites, parallel computation is commonly deployed; followed by summation in order to obtain a single tree score. However, basic arithmetic operations on IEEE 754 floating-point numbers, such as addition and multiplication, inherently introduce rounding errors. Consequently, the order in which floating-point operations are executed affects the exact result value, yielding IEEE 754 floating-point arithmetic non-associative. Moreover, parallel reduction algorithms in scientific codes re-associate operations based on the CPU core-count, resulting in different round-off errors. These low-level discrepancies can cause heuristic search algorithms to diverge and thereby propagate up to discrepancies in high-level results.

This effect has been observed in phylogenetics, sheet metal forming, climate and weather modeling, power grid analysis, atomic and molecular dynamics, as well as fluid dynamics. We observe that varying the degree of parallelism results in diverging phylogenetic tree searches (high level results) for over 31 % *out of* 10 130 *empirical datasets. More importantly,* 8 % *of these diverging datasets yield trees that are statistically significantly worse than the best found tree (AU-test,* $p < 0.05$*).*

To alleviate this, we introduce the ReproRed reduction algorithm, which yields bit-identical results under varying core-counts, by maintaining a fixed operation order that is independent of the communication pattern. ReproRed is thus applicable to all associative reduction operations – in contrast to competitors, which are confined to summation. This facilitates integration into the Message Passing Interface (MPI) wrapper library KaMPIng, allowing for arbitrary associative reduction operators, such

as C++ lambdas or function pointers. Our ReproRed reduction algorithm only exchanges the theoretical minimum number of messages, overlaps communication with computation, and utilizes fast base-cases for local reductions.
ReproRed is able to all-reduce (via a subsequent broadcast) 4.1×10^6 *operands across* 48 *to* 768 *cores in* 19.7 *to* 48.61 µs*, thereby exhibiting a slowdown of* 13 *to* 93 % *over a non-reproducible all-reduce algorithm and* 8 *to* 25 % *over an non-reproducible Reduce-Bcast algorithm. Further, our algorithm outperforms the state-of-the-art reproducible all-reduction algorithm ReproBLAS (offers summation only) beyond* 10 000 *elements per core. Finally, we develop a variant of the widely used phylogenetic inference tool RAxML-NG, which yields bit-reproducible results under varying core-counts, with a slowdown of only* 0 *to* 6.7 % *(median* 0.93 %*) on up to* 768 *cores.*
In summary, we re-assess non-reproducibility in phylogenetic inference and present the first bit-reproducible parallel phylogenetic inference tool. Further, we integrate our reduction algorithm into an open-source MPI-wrapper library, which facilitates the transition to bit-reproducible code.

5.1 Introduction

Next to disseminating results, scientific publications also aim to convince the reader of the results' validity [Mes10]. However, despite the fact that reproducibility is crucial for validating scientific claims [IT18], practical attempts to reproduce findings frequently fail (Section 2.10). Further, a failure to replicate a result might also indicate the inability to generalize the respective finding to variations in experimental conditions that were previously perceived as being irrelevant [GL19]. This observation induces a more nuanced picture of the corresponding scientific claim and paves the avenue for more specific inquiries. Thus, in order to avoid investigating measurement noise, computational results should be invariant (produce bit-identical outputs) under different CPU types and CPU core-count.

Moreover, bit-reproducibility is important for fulfilling contractual obligations, for example, in drug and nuclear reactor design [BK13], as well as for legal reasons [CLP16]. Bit-reproducibility also facilitates code debugging and verification [AFH14; DN15; JAM20; RRA11; Vil+09], as well as code extension by other researchers [SKC00]. Additionally, non-reproducible codes can follow different search paths, or converge more slowly, affecting running time measurements [Vil+09].

In computer science, the rounding errors that are inherent to floating-point computations can cause discrepancies in computed results as a consequence of, for example, differences in the available hardware instructions or number of Processing Elements (PEs; Section 2.10). This can lead optimization algorithms to pursue diverging search paths. Documented instances of these rounding errors impacting high-level outcomes span various scientific domains, including phylogenetics [DFS18; She+20a] (Section 2.10). Shen *et al.* [She+20b] found that varying the parallelization level affected the resulting tree topologies in 9 to 18 % of the empirical single gene datasets they examined (Section 5.2).

Common practices that aim to improve the reproducibility of computational results include documenting and archiving source code, input data, the manual steps to undertake, the software environment (e.g., the operating system), random seeds, and the used CPU model and number of PEs. However, the described supercomputer configuration or CPU model is unlikely to be available to future researchers that attempt to reproduce the findings. Thus, scientific codes should aim to yield bit-reproducible results across diverse CPU architectures and parallelization levels.

5.1.1 Contribution

In this work, we investigate the bit-reproducibility of phylogenetic tree inference across 10 130 empirical datasets under varying PE-count and Single Instruction Multiple Data Stream (SIMD) parallelization kernels. Our findings support that by Shen *et al.* [She+20b] in that 31 % of the tree searches diverge; with 8 % yielding significantly (AU-test, $p < 0.05$) worse phylogenetic trees due to differing levels of parallelism. To address this issue, we introduce a bit-reproducible version[1] of the phylogenetic tree inference tool RAxML-NG [Koz+19]. This bit-reproducible variant exhibits a slowdown of only 0 to 6.7 % (median 0.93 %) compared to the non-reproducible RAxML-NG version when benchmarked on 18 large, empirical datasets and on up to 768 cores.

Further, we introduce the distributed-memory parallel reduction algorithm ReproRed,[2] which enables bit-reproducible reductions for arbitrary associative reduction operations. This is contrast to competitors – who are confined to summation (Section 5.2) – and allows us to integrate ReproRed into the open-source MPI-wrapper library KaMPIng. ReproRed exchanges only the theoretical minimum number of messages, overlaps communication with computation, and utilizes a fast base-case for conducting local reductions. ReproRed is able to perform an all-reduction (via a subsequent broadcast) on 4.1×10^6 operands across 48 to 768 cores in 19.7 to 48.61 µs, thereby yielding in a slowdown of 13 to 93 % over a non-reproducible all-reduce algorithm and 8 to 25 % over a non-reproducible Reduce-Bcast algorithm. ReproRed outperforms the state-of-the-art reproducible all-reduction algorithm, ReproBLAS (summation only) for more than 10 000 elements per PE.

5.1.2 Outline of this Chapter

This chapter is organized as follows: Initially, we provide a comprehensive study on the reproducibility of phylogenetic tree searches under varying degrees of parallelism (Section 5.5.2) and discuss our results with respect to related work (Section 5.2.1). We then provide an overview of existing parallel reproducible (all-)reduction algorithms and non-reproducible baselines (Section 5.2). In Section 5.3.1, we outline the design of our bit-reproducible distributed reduction algorithm ReproRed, which supports arbitrary associative reduction operations. We discuss an efficient implementation and associated optimizations of ReproRed in Section 5.3.2. In Section 5.3.2, we discuss an all-reduce variant of ReproRed utilizing recursive-doubling, that halves the latency relative to a Reduce-Bcast approach. Next, in Section 5.4, we detail our bit-reproducible variant of the phylogenetic search tool RAxML-NG. We discuss the factors influencing ReproRed's running time in Section 5.5.3. In Section 5.5.1 we describe our experimental setup, and present our large-scale study on the reproducibility of phylogenetic tree searches in Section 5.5.2. The subsequent Section 5.5.4 provides an empirical evaluation of ReproRed's running time compared to the state-of-the-art bit-reproducible (all-)reduction algorithm ReproBLAS. Following this, we assess the running time penalty induced to RAxML-NG by using bit-reproducible reduction (Section 5.5.5), summarize our findings, and discuss directions of further research (Section 5.6).

[1] `https://doi.org/10.5281/zenodo.15017407`
[2] `https://doi.org/10.5281/zenodo.15004918`; LGPL

5.1.3 Parallel Reduction and All-Reduction

Even if we assume that the CPU executes identical instructions using identical floating-point CPU settings (Section 2.10), distributed-memory algorithms often use non-reproducible (all-)reduction operations [Rab00]. Here, the number of participating PEs affects the reduction order and ultimately the exact value of the end-result [Li+23; SMR24] (Section 2.10).

More specifically, given an input array of elements $E = [e_0, e_1, \ldots, e_{n-1}]$, distributed across p PEs, we desire to compute $r = e_0 \oplus e_1 \oplus \ldots \oplus e_{n-1}$, where $\oplus$ denotes a binary, associative operation such as summation or multiplication. In a distributed *reduction*, we return the result r to a single PE, while in an *all-reduction*, we return the result to all PEs. Conceptually, an all-reduction thus consists of a reduction followed by a broadcast. However, this approach results in a latency of $2 \cdot \log_2(p)$ messages: $\log_2(p)$ messages for the reduction phase and $\log_2(p)$ messages for the broadcast phase [San+19, ch. 13.2.1]. Dedicated all-reduction algorithms, such as recursive doubling as implemented in MPICH [BK13], exhibit a latency of only $\log_2(p)$ messages.

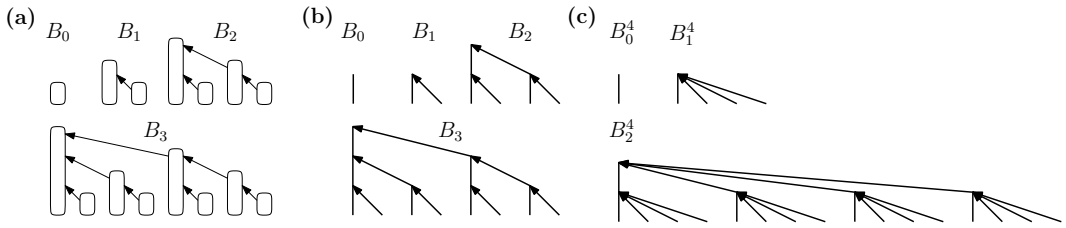


Figure 5.1 **(a)** Binomial trees B_0 to B_3. **(b)** Alternative representation of B_0 to B_3, indicating the messages sent in a binomial tree reduction. **(c)** B^4 binomial trees.

Binomial Tree Reduction For a small number of elements per PE, a binomial tree reduction is optimal in terms of message-count and size, as well as the number of operations performed, and thus, also running time [San+19, ch. 13.1.1]. Following Cormen *et al.* [Cor+01, ch. 19.1.1], we define a binomial tree as follows: The binomial tree B_0 of order 0 consists of a single node. A binomial tree B_k of order k comprises a root node whose children are the roots of the binomial trees of order $k-1, k-2, \ldots, 2, 1, 0$ (Figure 5.1.a). Further, we define a generalized binomial tree B^m, where the root of a binomial tree B_k^m has $m-1$ copies of each B_i^m for $i = k-1, k-2, \ldots, 2, 1, 0$ as children (Figure 5.1.b). Thus, in this notation, a canonical binomial tree is a B^2. In a binomial tree reduction, PEs send intermediate results along the edges of a binomial tree towards the root and also reduce the values in each step.

Non-Reproducibility of Distributed Reduction In High Performance Computing (HPC) applications, each PE typically handles multiple input elements. During the distributed reduction, we first reduce all local elements into a single intermediate result per PE. Next, we reduce these intermediate results via a binomial tree reduction [San+19, ch. 13.9] into a single global end result. However, in this algorithm, the PE-count influences the order by which we reduce the input elements, which in turn affect the rounding errors and consequently the value of the end result.

Further, topology-aware MPI implementations, such as MPICH, adjust the reduction tree's topology to accommodate the heterogeneous connectivity among PEs. Specifically, MPICH first reduces the elements of PEs on a single shared-memory machine before exchanging messages over the network [BK13]. Thus, for an identical PE-count, variations in topology

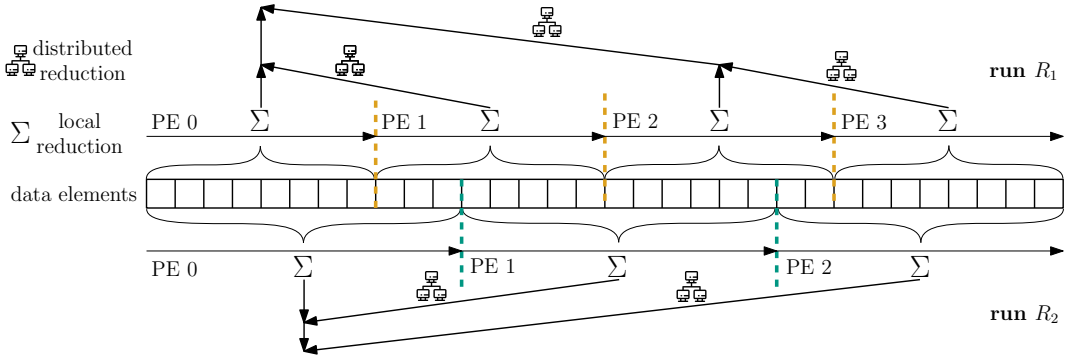


Figure 5.2 Common implementations of distributed-memory parallel reduction algorithms are not reproducible if the number of PEs differs between runs. First, each PE reduces its local elements, arriving at a single intermediate per-PE result. Next, these intermediate results are exchanged via messages over the network and further reduced into a single element. Note that the PE-count influences the reduction operation order and thus how results are rounded.

(e.g., cores per CPU or CPUs per compute node) can affect the reduction order and thereby the value of the end result [BK13].

Reduction in KaMPIng In Chapter 6, we present KaMPIng, our C++ wrapper around MPI, which includes the reduction and all-reduction collective operations. Here, the user can provide arbitrary associative reduction operations to KaMPIng's `reduce` and `allreduce` functions. Note that associativity is a necessary requirement for *parallel* reduction. Analogous to the standard KaMPIng `reduce`, our bit-reproducible reduction supports plain-MPI constants, function pointers, and C++ lambdas. For example, after initialization (Section 5.3.1), one could call our bit-reproducible reduction in KaMPIng as follows:

```
auto const result = comm.reproducible_reduce(
    send_buf(local_data), op([](auto& rhs, auto& lhs){ /* ... */ }));
```

5.2 Related Work

In this section, we first review related empirical studies on the non-reproducibility of phylogenetic inferences, and then also discuss competing reproducible reduction algorithms.

5.2.1 Non-Reproducibility of Phylogenetic Inference

Finding *the* most-likely phylogenetic tree among the super-exponentially large number of distinct possible phylogenies [SKK20] is $\mathcal{NP}$-hard [Roc06]. As a result, phylogenetic tree inference algorithms attempt to iteratively refine the currently best-known tree (Section 2.3). Consequently, minor differences in the Log-Likelihood (LLH) score of topologically identical intermediate trees can cause tree searches to diverge.

Darriba *et al.* [DFS18] where the first to report that Maximum-Likelihood phylogenetic tree searches can diverge due to floating-point inaccuracies (round-off errors). Following this observation, Shen *et al.* [She+20b] systematically analyzed 3515 single-gene datasets. They observed that variations in the CPU model, corresponding SIMD instruction sets, and the number of threads, yielded different phylogenetic trees for 9 to 18 % of empirical single-gene datasets. Furthermore, Shen *et al.* observed that 8.6 % of the non-reproducible phylogenetic

trees found by IQ-TREE and 25.21 % of the non-reproducible phylogenetic trees found by RAxML-NG where significantly worse than the respective best tree that was found ($p < 0.05$; AU-test [Shi02]).

5.2.2 Reproducible Reduction

(All-)reduction algorithms that yield bit-identical results under varying PE-counts have been studied for both, shared-memory [Vil+09] and distributed-memory [CLP17; Iak+15] systems.

Villa *et al.* [Vil+09] developed a bit-reproducible reduction algorithm on a shared memory Cray XMT supercomputer. They utilized this algorithm to sum over floating-point numbers using up to 16 PEs. Their algorithm ensures bit-reproducibility by fixing the reduction operation order independently of the PE-count. It therefore supports arbitrary associative reduction operations. Their *reduction* tree is based on a binomial tree, however, they accumulate $k > 2$ instead of 2 values sequentially at each inner node of the reduction tree, which reduces the required tree height. We do not compare our reduction algorithm to Villa *et al.* because the latter has been developed for a shared-memory Cray system which we do not have access to. Furthermore, our attempts to obtain the source code from the authors were unsuccessful.

For distributed-memory parallel machines, one has to consider explicit communication between PEs as well as scheduling the pair-wise reduction operation to PEs. Although the MPI standard encourages the implementation of bit-reproducible reduction operations [MPI4.1, ch. 6.9], existing MPI implementations prioritize performance over reproducibility [BK13]. Further, Balaji and Kimpe [BK13] claim that reproducible reduction cannot utilize knowledge about the concrete HPC topology for improving communication efficiency. However, we separate the communication pattern from the computation of the reduction, thereby demonstrating that this is not necessarily the case (Section 5.3.1).

Numerous bit-reproducible parallel *summation* algorithms exists [ADN20; AFH14; CLP16; Col+15; DN15; Li+23; Nea15]. These algorithms utilize variants of Kahan's compensated summation [Kah65; SMR24], which accumulates errors from each pair-wise summation in a separate variable. It thereby effectively increases the number of significant bits. Subsequent work by Demmel *et al.* [DN13] extends this concept to multiple variables, where each variable accumulates successively smaller parts of the error [SMR24]. However, these approaches are summation-specific and do not generalize to other reduction operations. Further, most of these approaches require a specific data representation and have a single target hardware architecture, which complicates code maintenance [SMR24].

The above algorithms have been implemented in three highly-optimized Basic Linear Algebra Subprograms (BLAS) libraries: ReproBLAS [ADN20], RareBLAS [CLP17], and ExBLAS [Iak+15]. However, RareBLAS [CLP17] targets parallel shared-memory machines, yielding it inapplicable to our distributed-memory context. Further, Lei *et al.* [Lei+23] show that ReproBLAS is consistently faster than ExBLAS. Consequently, we omit ExBLAS and only consider ReproBLAS as the state-of-the-art implementation for evaluating our ReproRed algorithm.

The IntelMPI and MPICH MPI implementations (Section 2.6) support reproducible reduction [BK13; Int25c]. However, both require a consistent PE-count between runs; which substantially simplifies the problem.

Further Baselines With *Gather-Bcast*, we denote an algorithm first described by He and Ding [HD01]: Initially, the algorithm gathers all input data elements on a single root PE. This root PE subsequently locally reduces the data elements in a fixed order, for instance,

from left to right. Finally, the root PE broadcasts the reduced result to all other PEs. The MPI standard recommends this algorithm for obtaining bit-reproducible results under varying PE-counts [MPI4.1, ch 6.9]. However, this approach incurs a bottleneck communication volume of $\mathcal{O}(p)$ messages and $\mathcal{O}(n)$ received elements at the root PE, where p is the number of PEs, and n is the number of input data elements. Further, it lacks parallelism, resulting in a running time of $\mathcal{O}(n)$. Moreover, the root PE requires sufficient memory to hold *all* input data elements – if not implemented using a streaming algorithm.

We also compare ReproRed against two *non-reproducible* baselines: A PE-local reduction using C++'s `std::reduce` followed by either IntelMPI's `MPI_Allreduce`, or a `MPI_Reduce` and a subsequent `MPI_Bcast`. We denote these algorithms as *Allreduce* and *Reduce-Bcast*, respectively. Here, IntelMPI selects the "best" (all-)reduction algorithms by using yet undisclosed heuristics. We attempted to re-configure IntelMPI to use a binomial tree broadcast, binomial tree reduction, and recursive doubling all-reduction by adjusting the respective `I_MPI_ADJUST` variables. However, this resulted in performance degradation, possibly because of the loss of topology-aware optimizations. We therefore perform all benchmarks with IntelMPI's default settings.

5.3 ReproRed: Operation-Agnostic Bit-Reproducible Parallel Reduction

The only approach for ensuring bit-reproducibility while only assuming the associativity of the underlying reduction operation, is to maintain a fixed reduction operation order [AFH14]. As parallel reduction typically represents only a single step in complex numerical algorithms, optimizing the load-balancer for efficient reduction operations may induce upstream work imbalances, potentially undermining the running time benefits of a more efficient reduction algorithm. Therefore, we design ReproRed to maintain a fixed reduction operation order independently of the communication pattern and do not impose a specific data distribution.

5.3.1 Design of ReproRed

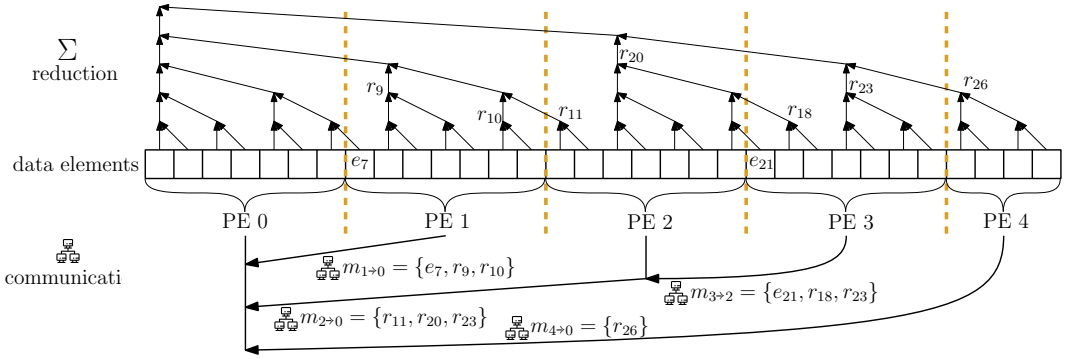


Figure 5.3 Decoupling the order of reduction operations from communication. In ReproRed, the input data elements are always reduced in an order determined by a binary tree, regardless of the number and topology of the PEs. We implement the communication among the p PEs via a binomial tree, which requires the minimum number of messages $\mathcal{O}(p-1)$.

The core idea of ReproRed is to decouple the communication among PEs from the order by which we apply reduction operations, allowing us to maintain a fixed operation order under varying PE-counts (Figure 5.3). To this end, we organize the reduction operations as a binary tree with the input data elements being the leaves. This *pair-wise reduction* also

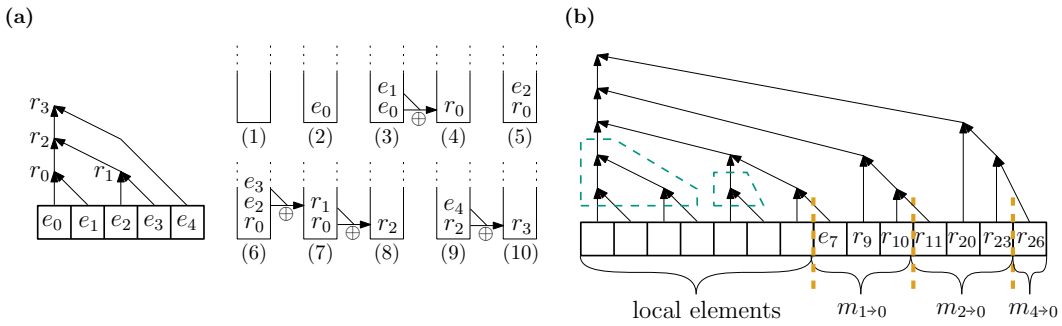


Figure 5.4 Implementation of ReproRed. **(a)** A post-order traversal of a binary tree (left) can be implemented using a stack (right). When we encounter a leaf node during the traversal, we push the respective data element onto the stack. For a non-leaf node, we pop the two topmost elements from the stack, reduce them, and push the result back onto the stack. Ultimately, at the root of the tree, the remaining stack element contains the final result of the post-order traversal. **(b)** Reduction operations performed on PE 1 in Figure 5.3. The received messages $m_{i \to j}$ are consecutively stored after the local elements. Note that some received elements already constitute intermediate results of the overall reduction. Further, the PE can reduce most of its local elements (green, dashed boxes) thereby overlapping computation with communication.

reduces the overall rounding errors [Hig93]. Further, we orchestrate communication via a binomial tree, which is theoretically optimal for small message sizes [San+19, ch. 13.9], and used, for instance, by MPICH [BK13]. Note that this design implies that PEs can generally not reduce all their local elements into a single intermediate result, but are required to send the intermediate results of multiple subtrees to their parent PE in the communication tree instead. However, we expect the startup overhead for each message to exceed the cost of transmitting the respective additional floating-point values. We explore the efficient implementation of this design in the next paragraph and discuss its implications on running time in Sections 5.5.3 and 5.5.4.

Post-Order Traversal via Stack Operations We can implement a post-order traversal of an arbitrary binary tree via a stack S (Figure 5.4.a). Let $E = (e_0, e_1, \ldots, e_{n-1})$ be the *ordered* sequence of elements to be reduced. In the reduction tree, each *leaf* (in-degree 0) corresponds to an element $e_i \in E$. During the post-order traversal, upon encountering a leaf, we push the corresponding element onto S. We denote this as *addValue*. Conversely, each *non-leaf* node of the reduction tree represents an intermediate result r_j, which we index in post-order. When we reach a non-leaf node, we *pop* the two topmost elements from S, apply the reduction operator $\oplus$, and *push* the result back onto S. We denote this as *reduceTwoValues*. Ultimately, the root of the reduction tree (out-degree 0) contains the final result of the reduction; and it will be the sole element remaining on the stack. Consequently, we can implicitly represent the reduction tree through a sequence of *addValue* and *reduceTwoValues* operations.

Distributing the Post-Order Traversal Among the PEs To parallelize the reduction, we distribute this sequence of *addValue* and *reduceTwoValues* operations, which describe the post-order traversal of the reduction tree, across all PEs. For this, we assign each *addValue* operation to the PE that holds the respective data element. Each *reduceTwoValues* operation corresponds to a specific non-leaf node in the reduction tree. This node, including its descendants, induces a subtree in the reduction tree, with a set of (consecutive) input data elements as leaf nodes. Each of these elements resides on a specific PE. The set PEs on which

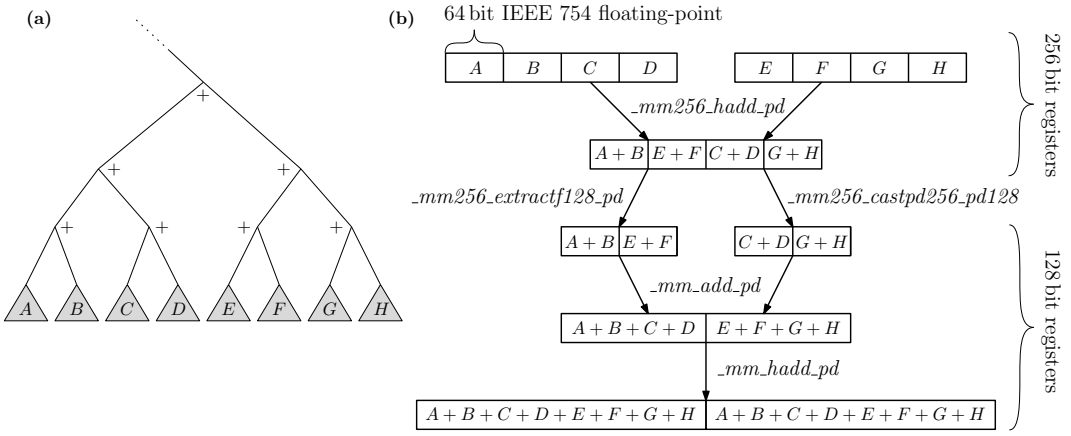


Figure 5.5 Pair-wise addition of eight 64 bit IEEE 754 floating-point values using AVX2 vectorization. **(a)** The subtree of the reduction tree to be reduced. $A, \ldots, H$ can be either input data elements or intermediate results from lower subtrees. **(b)** Implementation of the AVX2-vectorized reduction of the subtree depicted in (a). Note that the order of operations is consistent with a non-vectorized pair-wise addition implemented as a post-order traversal as described in Figure 5.4.

the elements of a reduction tree subtree reside have a Lowest Common Ancestor (LCA) in the communication tree. We assign the *reduceTwoValues* operation to this LCA-PE.

We determine this assignment of operations to PEs during a pre-processing step, which we run once for each data distribution. Thus, when one invokes the distributed reduce operation with concrete values – possibly thousands of times per second (Section 5.5.5) – each PE already knows all operations it needs to execute a priori (Figure 5.4.b).

During a reduction, each PE initially triggers the asynchronous message-receive operations, thereby allowing computation to overlap with communication. As each PE knows the number of messages and data elements it will receive, it can arrange local and received data in consecutive memory locations. The operations of a PE define reduction operations on a subtree of the overall reduction tree – with input data elements and intermediate results being computed on other PEs as leaves. The PE then starts reducing its local subtree. Here, we apply a fast base-case for fully local subtrees; for example, we parallelize summation using manually optimized AVX2 code (see Section 5.3.2). If a PE needs to perform a *reduceTwoValues* operation for which one operand has not been received yet, the PE blocks until the respective message exchange has been completed. Upon completing its reductions, the PE sends the computed intermediate results to its parent in the communication tree. Note that the stack will already contain the elements ordered as expected from the receiving PE. If an all-reduction shall be executed, we subsequently broadcast the final result from the reduction's root-PE.

5.3.2 Optimizations

The three key optimizations we implement are to overlap computation with communication, a fast base-case for reducing PE-local data elements, and to communicate via a B^4 instead of a B^2 binomial reduction tree (Figure 5.1).

Vectorized Reduction of PE-local Subtrees The peak floating-point operations per second (FLOPS) in CPUs increases substantially when exploiting SIMD capabilities [Dol18]. Importantly, our binary reduction tree allows for parallelizing the reduction without altering

the order of operations. In contrast, a left-to-right reduction is inherently sequential and requires vectorized algorithms to change the operation order, thereby compromising bit-reproducibility. As an example, we implement an AVX2-vectorized base-case for summing over PE-local subtrees in the reduction tree (Figure 5.4). Note that analogous, fast base-case implementations are feasible for arbitrary reduction operations.

Our AVX2 implementation reduces a subtree comprising eight elements or intermediate results using two 256 bit registers, two 128 bit registers, two load operations, and five additional SIMD instructions (Figure 5.5). We apply the reduction recursively, that is, the leaves of this eight-element tree may be intermediate results of preceding vectorized reductions. For example, to reduce 512 input elements, we execute three levels of AVX2-optimized reductions, resulting in 73 invocations of the 8-value AVX2 kernel (Figure 5.5.b).

First, we horizontally sum pairs of elements in two 256 bit registers and combine the results into a single 256 bit register (Figure 5.5). Since AVX2 instructions operate within 128 bit lanes, we must separately extract the upper and lower 128 bit segments after the initial horizontal addition in order to maintain the correct summation order. Subsequently, we conduct an element-wise addition on the two resulting 128 bit registers, followed by horizontal addition on the resulting 128 bit register. Again, as the operation order and thus, the round-off errors remain unaltered, a vectorization with different SIMD register widths will produce identical results.[3]

Compilers, such as GCC, attempt to automatically vectorize code. However, a reduction with manually optimized AVX2 vectorization achieved a speedup of 2.9 % (median) compared to the compiler-vectorized reference on the datasets in Section 5.5.4. For datasets with tens of millions of elements across all PEs, we attain speedups of up to 200 %.

Using a B^4 Reduction Tree We implement the communication tree as a B^4, instead of a B^2 tree (Figure 5.1, not shown in all figures). We empirically determined this appropriate degree of $m = 4$ and gain a speedup of 5.6 % (median; data not shown) over a B^2 tree. This acceleration is attained because the algorithm is able to reduce data elements at lower levels of the communication tree, thereby reducing message sizes and decreasing the number of times an element will be forwarded (Section 5.5.3).

Overlapping Communication with Computation We implement ReproRed such as to overlap computation with communication (Figure 5.4) by utilizing non-blocking MPI messages (Sections 2.6 and 6.4.5). Specifically, each PE first posts the MPI receive operations before reducing its local elements. A PE blocks and waits for a message only when it requires an element that has not yet been received. By overlapping computation with communication, we accelerate the reduction by 4.4 % (median; data not shown).

Bit-Reproducible All-Reduction using Recursive Doubling In ReproRed, all-reduction is currently implemented as a reduction followed by a broadcast (Figure 5.6.a). This induces a latency of $2\log_2(p)$ messages for p PEs. Here, we propose a variant with a latency of $\log_2(p)$ messages using the recursive doubling algorithm [TRG05]. In the first step of recursive doubling, PEs with a distance of one exchange data. In a second step, PEs with a distance of two exchange data – including the data they received from the other PE in the first step. In subsequent steps, the distance increases exponentially, until, after $\log_2(p)$ steps, all PEs

[3] We experimentally verified that a SSE3 implementations yields bit-identical results with our AVX2 implementation.

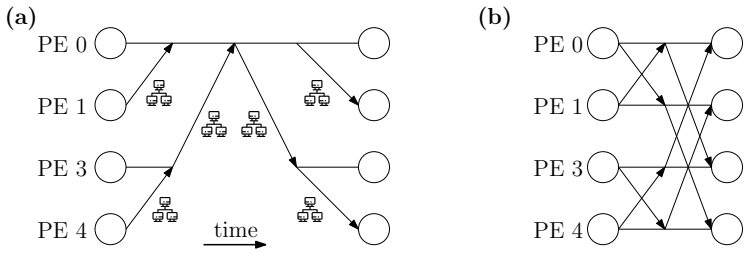


Figure 5.6 (a) The ReproRed-TwoPhase all-reduction algorithm consists of a binomial tree ReproRed reduction, followed by a binomial tree broadcast. **(b)** The ReproRed-RecursiveDoubling all-reduction algorithm is based on the recursive doubling all-reduction [TRG05]. It reduces the latency from $2\log_2(p)$ to $\log_2(p)$ for p PEs compared to ReproRed-TwoPhase.

know each other PEs' data – or a reduced form thereof. Note that in recursive doubling, a PE always has knowledge of elements that reside on PEs that are *consecutive* in PE-id (*rank*)-space, thus allowing for the reduction of intermediate results (Figure 5.6.b). Future work will include implementing this ReproRed-RecursiveDoubling algorithm.

5.4 Reproducible Phylogenetic Inference

We present Repro-RAxML-NG, a bit-reproducible variant of the widely used phylogenetic tree inference tool RAxML-NG (developed by our lab). For this, we evaluate different bit-reproducible all-reduction implementations and discuss the observed slowdowns compared to the standard non-reproducible RAxML-NG version in Section 5.5.5. Repro-RAxML-NG currently supports bit-reproducible tree evaluation and tree searches (Section 2.3). At present, the pattern compression, tip inner, and site repeats optimizations as well as the vectorization of phylogenetic likelihood derivatives required for branch length optimizations, are disabled to keep code changes to the necessary minimum for our experiments. However, these simplifications do not affect reproducibility for we plan to reintroduce them to Repro-RAxML-NG. We empirically verify that Repro-RAxML-NG yields bit-identical results across all configurations when run on the empirical datasets described in Section 5.5.1 – some of which are known to cause diverging tree searches under varying PE-counts.

5.5 Evaluation

In the following section, we first discuss the factors influencing the running time of ReproRed in comparison to the state-of-the-art bit-reproducible reduction algorithm for summation, ReproBLAS. Next, we empirically evaluate the running times of both algorithms, along with non-reproducible baselines in an isolated reduction benchmark (Section 5.5.4). In Section 5.5.5, we report on the overhead incurred in RAxML-NG when ensuring the results are bit-reproducible.

5.5.1 Experimental Setup

We conduct our experiments on the SuperMUC-NG HPC system.[4] Each compute node is equipped with 96 GiB of memory and comprises two Intel Skylake Xeon Platinum 8174

[4] `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`

processors with 24 cores, each running at a fixed 2.3 GHz. The nodes communicate via an OmniPath network with a fat tree topology and a bandwidth of 100 Gbit s^{-1}. The operating system is SUSE Linux Enterprise Server 15 SP3 running Linux Kernel version `5.3.18-150300.59.63`. We compile our benchmark applications using GCC version 12.2.0 with full optimizations enabled (`-O3`) and all assertions in RAxML-NG being disabled; we employ IntelMPI `2021.9.0`. Repro-RAxML-NG[5], our reproducible variant of RAxML-NG, is based on commit `816f9d13`,[6] which we use as a reference.

We assess the computational reproducibility on 10 130 empirical phylogenetic datasets from TreeBASE [Pie+09] comprising more than 100 000 distinct taxa collected from over 4000 publications. These TreeBASE datasets contain 4 to 4019 taxa, 1 to 179 partitions, and 21 to 3 848 295 sites – with up to 214 492 unique columns in the respective Multiple Sequence Alignments (Section 2.2). For our running time benchmarks, we utilize the same datasets as previously used to evaluate the performance of RAxML-NG upon its release [Koz+19]. These datasets comprise between 1371 and 21 410 970 sites. The largest are thus suitable for parallel inference on dozens or hundreds of PEs, where we anticipate our approach to perform the worst due to the extended critical path (Section 5.5.3).

5.5.2 Reproducibility of Phylogenetic Inference

For each TreeBASE [Pie+09] dataset (Section 5.5.1), we execute the same RAxML-NG [Koz+19] application binary eight times, using identical hardware, search settings, and random seeds; and only vary the degree of parallelization. Specifically, we conduct inferences using one to five PEs with the AVX2 vectorization, as well as single-PE inferences with SSE3, AVX, AVX2, and compiler auto-vectorization. We quantify the divergence between the resulting trees using the relative Robinson-Foulds distance [RF81] (Figure 5.7). Additionally, we assess the statistical significance of differences in tree likelihoods via the Approximately Unbiased (AU) test (Figure 5.8).

Robinson-Foulds Distance Let a *bipartition* refer to the split of a tree's tips into two disjoint sets. Each edge of a tree induces a bipartition, and collectively, the set of all bipartitions fully describes a tree. The Robinson-Foulds distance [RF81] quantifies the proportion of bipartitions that differ between two trees.

Approximately-Unbiased test When multiple trees exhibit sufficiently similar likelihood scores, there is no compelling reason to favor one over the others. Thus, following Shen *et al.* [She+20b], we apply the AU-test [Shi02] to determine if two tree topologies have significantly different likelihood scores ($p < 0.05$). If the difference is significant, this indicates that the tree search conducted under a particular level of parallelism yielded a significantly worse tree than one conduced under a different level of parallelism. To prevent erroneous rejection of the correct tree in sets with similar likelihoods [Shi02], we consider trees with a log-likelihood difference of less than 10^{-3} to not be significantly different.[7]

Results We consider a dataset as *diverging* if the phylogenetic inferences conducted on it yield at least two topologically distinct trees. Among the 10 130 TreeBASE [Pie+09]

[5] `https://doi.org/10.5281/zenodo.15017407`
[6] `https://github.com/amkozlov/raxml-ng/commit/816f9d13`
[7] Shimodaira's implementation of the AU-test generated the following warnings: "small variance", "theory does not fit well", and "regression degenerated".

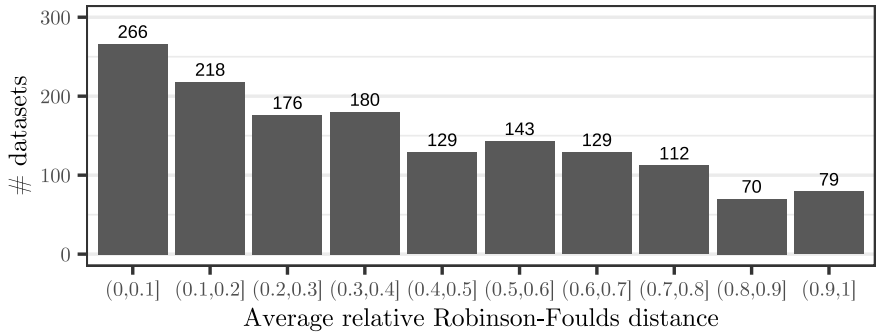


Figure 5.7 Average relative Robinson-Foulds distance between all distinct tree topologies resulting from diverging tree searches because of different degrees of parallelization. Only datasets with at least two distinct tree topologies (31 % of all datasets) are shown.

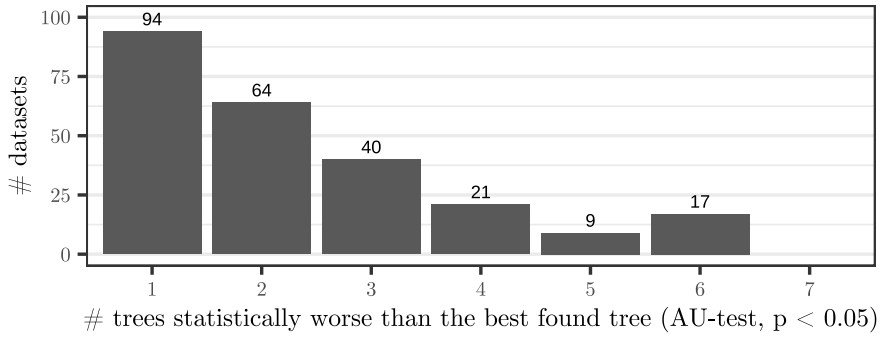


Figure 5.8 Number of trees found in tree searches diverging due to varying degree of parallelization that are significantly worse than the best found tree. We show only the 8 % of diverging datasets where the tree searches yielded at least one tree that is significantly worse than the best found tree (AU-test, $p < 0.05$).

datasets (Section 5.5.1) where the phylogenetic inference completed without error, 31 % (3111) datasets produced multiple topologically distinct trees (Figure 5.7). Of these 3111 datasets with diverged inferences, 46 % (1418) exhibit a log-likelihood difference exceeding 10^{-3} log-likelihood units. Further, 8 % (245) contain significantly different trees (AU-test, $p < 0.05$) in the resulting tree set (Figure 5.8).

Compared to Shen *et al.* [She+20a], who analyzed 3515 single-gene datasets, we observe a higher rate of diverging tree searches (31 % vs. 9.3 % with RAxML-NG) yet fewer significant differences (8 % vs. 25 % of diverged tree searches). The increased proportion of diverging tree searches in our analysis likely arises because we assess eight distinct degrees of parallelization, instead of two, as in Shen *et al.* [She+20a]. Further, we classify trees with a LLH-difference $< 10^{-3}$ as *not being statistically different* (see above), while Shen *et al.* [She+20a] do not apply this (arbitrary) threshold. If we omit this filtering step, 38 % (1186) of our datasets with diverging tree searches yield significantly different trees (AU-test, $p < 0.05$).

Conclusion In conclusion, we demonstrate that phylogenetic tree search is vulnerable to different rounding errors – caused by varying degrees of parallelization – to induce divergent tree searches. This finding supports the results of Shen *et al.* [She+20b]. Note that divergences

in trees due to parallelization differ in their distribution from those caused by variations in random seeds or bootstrapping [Efr79; Fel85]. More importantly, differing degrees of parallelization introduce an unknown bias in the distribution of resulting trees. In contrast to this, the effects and generalization properties of varying random seeds and employing bootstrapping are well-studied [EHH96; Hed92; San89]. Whether differences between trees that are considered as being not significantly different from each other also have implications on their biological interpretation is non-trivial to answer and remains an open question. Therefore, we advocate for a bit-reproducible implementation of phylogenetic tree searches.

5.5.3 Factors Impacting ReproRed's Running Time

In the following, we discuss the factors impacting the running time of ReproRed compared to ReproBLAS: The number and size of messages exchanged, the local work, as well as the length of the critical path.

Message Size And Count For a *reduction*, each of the p PEs, except the root, has to send at least one message in order for its data elements to be included in the end result. This yields the lower bound of $p-1$ on the message-count [San+19, ch. 13.9]. ReproRed sends exactly $p-1$ messages along the edges of a binomial tree. This also holds for the B^4 tree described in Section 5.3.2, as again, each PE, except the root, sends exactly one message. ReproBLAS (Section 5.2) can utilize an arbitrary communication pattern, for instance, a binomial tree, which exchanges $p-1$ messages. Further, for an *all-reduction*, the two-phase approaches ReproRed-TwoPhase and ReproBLAS-TwoPhase (as well as Reduce-Bcast) exchange $2(p-1)$ messages. We experimentally verify in our benchmarks that the implementations of these algorithms exchange exactly the described number of messages.

The message sizes exchanged do not differ substantially between ReproBLAS and ReproRed. ReproBLAS exchanges six double precision floating-point values, that is, 384 bit, per message [ADN20] – which we experimentally verify for our benchmarks using the Score-P profiler [Knü+12]. Conversely, the message size in ReproRed varies depending on to the number of elements, the number of PEs, and the distribution of the elements among the PEs.

Consider, a distribution of n elements e_i ($i \in \mathbb{N}$) among p PEs, such that each PE holds the same power-of-two number of elements (Figure 5.9.a). Specifically, PE j holds elements $e_{j\cdot 2^k}, \ldots, e_{(j+1)\cdot 2^k-1}$, where $k = log_2(n/p)$. We denote a pair (e_i, e_{i+1}) as a *PE-boundary* if e_i and e_{i+1} are held by different PEs. Under this data distribution, PE-boundaries are located in-between full subtrees of the binary reduction tree. Thus, each PE is able to reduce all elements it holds into a single intermediate result. Further, during communication, the PE can immediately reduce the incoming values with its local intermediate results. Consequently, each message has a length of exactly one intermediate result (64 bit).

In contrast, consider the distribution in Figure 5.9.b: The first PE holds one element more than a power-of-two, while all other PE still hold a power-of-two elements.[8] Specifically, PE 0 holds e_0, e_{2^k+1} and a PE $j > 1$ holds the elements $e_{j\cdot 2^k+1}, \ldots, e_{(j+1)\cdot 2^k}$. Under this distribution, the PE-boundaries cut $\log_2(n)$ edges of the binary reduction tree. Each cut edge represents an element or intermediate result that must be exchanged across the network. Moreover, the two PEs of a boundary might not share an edge in the communication tree. In such cases, the corresponding data must be sent up the communication tree until they reach the lowest common ancestor of the two PEs at the boundary.

[8] The last PE has one element less, which we omit in the following discussion for simplification.

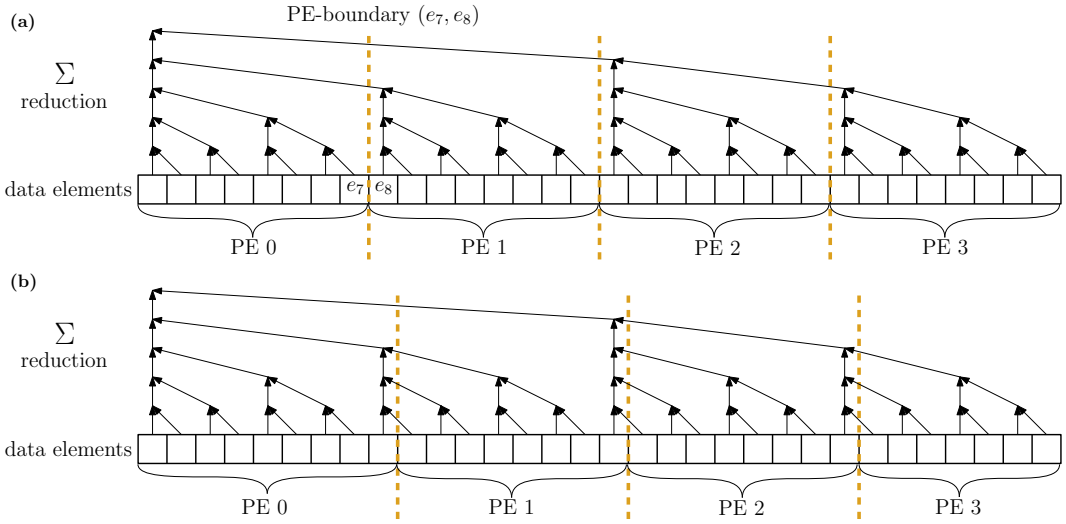


Figure 5.9 Examples of the best (a) and worst (b) case data distributions with regard to the message size (and critical path length) in ReproRed. Dashed lines represent PE-boundaries. **(a)** In the best case, each PE holds a power-of-two number of elements and is thus able to reduce its elements into a single intermediate result. Here, each PE-boundary cuts exactly one edge of the reduction tree. **(b)** In the worst case, the PE-boundaries cut $\mathcal{O}(\log_2(n))$ edges of the reduction tree. Each cut edge corresponds to an intermediate results that has to be exchanged over the network.

In summary, each PE-boundary cuts up to $\log_2(n)$ edges of the reduction tree, resulting in the respective PE to send this number of elements to its parent-PE in the communication tree. Further, each of these elements is relayed up to $\log_2(p)$ times. We observe that the messages exchanged in our isolated benchmarks (Section 5.5.4) contain 1 up to 24 (median and mean 11) elements, each comprising 64 bit. Thus, we expect the startup overhead for each message to substantially exceed the cost of transferring these additional bytes [San+19, ch. 13.6.2].

Local Work ReproBLAS executes 9 floating-point and 3 bit-wise operations per floating-point element in the input [ADN20], regardless of the data distribution. In contrast, ReproRed requires five floating-point SIMD operations to sum eight fully-local floating-point values. Further, for each pair-wise reduction involving received data, ReproRed must increment an iterator, extract a bit from a bit-vector, and perform a conditional jump in order to decode the operation (Section 5.3.1). Additionally, for each non-local reduction (`reduceTwoValues`) operation, ReproRed performs two `pop` and one `push` operation on a stack.

Using the Score-P profiler [Knü+12], we experimentally obtain that ReproBLAS requires more running time for its local work compared to ReproRed in our isolated benchmarks (Section 5.5.4). The precise overhead for local work of ReproBLAS over ReproRed depends on the number of elements per PE and varies by up to a factor of two (data not shown).

Length of Critical Path During Message Exchange We define an algorithm's *critical path* as the longest task sequence where each task depends on the preceding one. These tasks include message exchanges and local operations, such as applying a reduction operation to two elements. For both, ReproRed and ReproBLAS, the number of messages on the critical path in a *reduction* corresponds to the longest path in a binomial tree, that is $\log_2(p)$ [TRG05], where p is the number of PEs. ReproBLAS allows the PEs to reduce

all their local elements into a single intermediate result in parallel. These elements are subsequently reduced via at least $\log_2(p)$ steps, using, for example, a binomial tree reduction. In each step, ReproBLAS executes 9 floating-point operations and 3 bit-operations [ADN20]. Thus, ReproBLAS executes $\mathcal{O}(n/p + \log_2(p))$ operations on its critical path. In ReproRed, depending on the data distribution, PEs cannot reduce their local elements into a single intermediate results, but rather a (small) set of intermediate results. As each PE-boundary cuts up to $\log_2(n)$ edges of the reduction tree, the respective number of operations is thus located on the critical path. Overall, assuming $n > p$, there are $\mathcal{O}(n/p + \log_2(n))$ pair-wise reductions – and thus arithmetic operations – on ReproRed's critical path. Note that the critical paths are of the same length asymptotically, but differ in a constant factor. We implement ReproRed to overlap communication with computation, thereby mitigating the impact of this extended critical path (Section 5.3.2).

During an *all-reduction*, the two-phase approaches, ReproRed-TwoPhase, ReproBLAS-TwoPhase, and Reduce-Bcast, perform a binomial tree reduction followed by a binomial tree broadcast. Thus, these methods exchange $2\log_2(p)$ messages on the critical path. However, all-reduce algorithms with only $\log_2(p)$ messages on their critical path exists, such as recursive doubling [TRG05]. We outline a recursive doubling based implementation of ReproRed in Section 5.3.2. Thus, when assessing ReproRed's *reduction* performance, one can argue for using Reduce-Bcast as the reference benchmark.

5.5.4 Isolated Bit-Reproducible Reduction

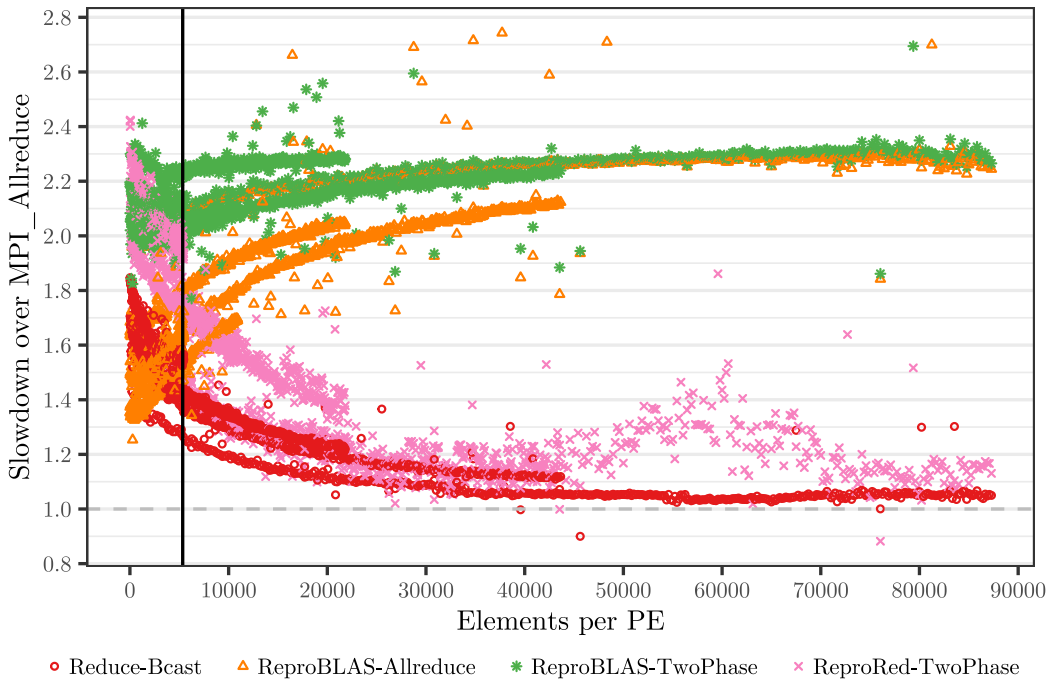


Figure 5.10 Slowdown of bit-reproducible all-reduce algorithms over an non-reproducible all-reduction. Each point represents the median runtime of 4000 iterations for this combination of the number of elements and number of PEs divided by the respective median runtime of the non-reproducible reference (`MPI_Allreduce`) The vertical line represents the median number of LLH-values per PE in our analyses of the real-world datasets (Section 5.5.5).

We compare the running times of the bit-reproducible *all-reduce* algorithms, ReproBLAS (state-of-the-art), ReproRed (ours), and Gather-Bcast (recommended by the MPI-standard), as well as the non-reproducible `MPI_Allreduce` and Reduce-Bcast (Section 5.2). As only ReproRed supports reduction operators beyond summation, we use summation as the reduction operator in our benchmarks.

The actual data distribution, and in particular the number of elements per PE, impact the running time of ReproRed. Thus, we utilize the same number of elements (i.e., Log-Likelihood values; Section 2.1), as in our phylogenetic inferences (Section 5.4). We distribute 1000 to 4 190 000 double precision (64 bit) IEEE 754 floating-point elements across 48 to 768 PEs, maintaining approximately equal numbers of elements per PE. This configuration reflects realistic usage scenarios for RAxML-NG.

We run 5000 iterations for each data distribution, discarding the first 1000 runs for each distribution to warm up the network caches. The absolute runtimes for `MPI_Allreduce` range from 1.6 to 600 µs (median 10.8 µs). We verify that the reductions performed with ReproBLAS and ReproRed yield bit-identical results across different data distributions and PE-counts. Additionally, we verify that the results of all algorithms differ by less than 10^{-6}.

Gather-Bcast (Section 5.2) is 9.8 to 125 (median 31) times slower than `MPI_Allreduce`. In addition, Gather-Bcast necessitates at least one of the PEs to have sufficient memory to store *all* elements. We thus consider Gather-Bcast impractical and omit it in Figure 5.10.

Further, ReproRed substantially outperforms the algorithm we describe in previous work [Ste22] (data not shown). Profiling results obtained with Score-P [Knü+12] (data not shown), indicate that this improvement results from minimizing the message-count to the theoretical minimum of $p - 1$. We thus omit this initial algorithm in the further discussion.

ReproRed is currently implemented as a *reduction* algorithm. We implement the ReproRed-TwoPhase *all-reduction* as a ReproRed reduction and a subsequent `MPI_Bcast`. This approach doubles the number of messages on the critical path compared to an all-reduction algorithm using recursive doubling from $\log_2(p)$ to $2\log_2(p)$ [Rue+21] (Section 5.5.3). We outline a recursive doubling based ReproRed variant Section 5.3.2, but do not implement it (yet). To evaluate the overhead induced by this two-phase strategy, we measured the performance of a non-reproducible local reduction followed by an `MPI_Reduce` and an `MPI_Bcast` (denoted as *Reduce-Bcast*; Figure 5.10). The median number of elements per PE in our phylogenetic experiments (Section 5.4) is 1935. For such a data distribution, Reduce-Bcast is 61 % slower (median) than all-reduce. However, for more than 45 000 elements per PE, Reduce-Bcast is merely 5 % slower (median; Figure 5.10).

We find that ReproRed is faster than ReproBLAS-TwoPhase under most data distributions with more than 3000 elements per PE, and faster than ReproBLAS-Allreduce when more than 10 000 elements are stored on each PE (Figure 5.10). As detailed in Section 5.5.3, the running times of ReproBLAS and ReproRed depend on the amount of local work, the message-count on the critical path and, for ReproRed, also the data distribution. While ReproRed performs less local work than ReproBLAS, it also increases the number of computations on the critical path (Section 5.5.3). In accordance to this, we observe that ReproRed is slower than ReproBLAS for less than a few thousand elements per PE. However, as the number of elements per PE increases, local computation begins to compensate these effects. This explains ReproRed's notably faster performance on data distributions with tens of thousands of elements per PE (Figure 5.10). For instance, ReproRed is 37 % (median) times faster than ReproBLAS on 8000 to 12 000 elements per PE and 92 % (median) times faster on more than 30 000 elements. Note, however, that ReproBLAS utilizes additional local work to guarantee the accuracy of the final result. In contrast, ReproRed's pair-wise summa-

tion merely increases the accuracy [Hig93]. Further, ReproBLAS is fundamentally confined to summation as the reduction operator, while ReproRed supports arbitrary associative reduction operators.

5.5.5 Reproducible Phylogenetic Tree Search

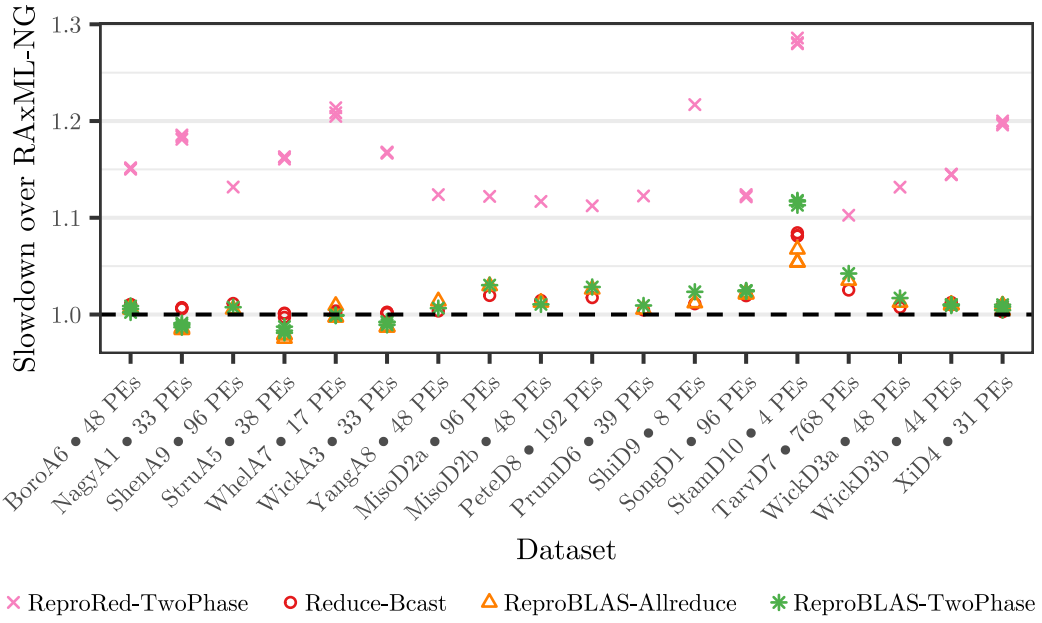


Figure 5.11 Slowdown of bit-reproducible Repro-RAxML-NG over reference RAxML-NG. Reduce-Bcast uses a non-reproducible reduction but includes the remaining modifications required for bit-reproducibility. The variants ReproBLAS and ReproRed yield bit-reproducible results.

We assess the slowdown of our bit-reproducible Repro-RAxML-NG (Section 5.4) in comparison to the unmodified reference RAxML-NG (Figure 5.11). Specifically, we evaluate Repro-RAxML-NG with the reproducible all-reduce algorithms ReproBLAS (state-of-the-art), Gather-Bcast (recommended by the MPI-standard), and ReproRed (ours). Further, we present results for the non-reproducible `MPI_Allreduce`, and TwoPhase approaches (for a rationale, see Section 5.5.3). We utilize the PE-counts recommended by RAxML-NG for the respective datasets, resulting in 253 to 11 903 (median 5569) unique Multiple Sequence Alignment columns per PE and 1935 (median) all-reduce operations per second (for the RAxML-NG reference version). For consistency, we disable optimization features not supporting bit-reproducibility in all variants (Section 5.4). As only experiments where RAxML-NG followed identical tree search paths are comparable, we exclude datasets where the variants did not converge on the same topologically identical best tree. Further, we verify that the resulting tree LLHs (Section 2.3) differ by less than 10^{-3} LLH-units among all variants.

Repro-RAxML-NG using the Gather-Bcast all-reduce algorithm is 21 to 386 % (median 26 %) slower than the reference. It consistently is the slowest variant across all configurations; we hence omit it in further discussion. Additionally, since ReproRed is never slower than the approach we presented in previous work [Ste22], we also omit the latter.

The Repro-RAxML-NG variant using ReproRed-TwoPhase is 10 to 29 % (median 17 % slower than the reference. When comparing ReproRed-TwoPhase to Reduce-Bcast, we

attribute up to 30 % (median 1 %) of this slowdown to the adoption of a two-phase approach (Section 5.2). Consequently, we attribute the residual slowdown to ReproRed's extended critical path (Section 5.5.3). Given that there are 1935 (median) all-reduce operations per second, even a minor running time increase of a single all-reduce operation substantially affects the overall running time.

Utilizing the bit-reproducible ReproBLAS all-reduction, Repro-RAxML-NG only exhibits a 0 to 6.7 % slowdown (median 0.93 %) compared to the non-reproducible reference. This is consistent with the isolated measurements described in Section 5.5.4, where ReproBLAS outperforms ReproRed-TwoPhase on less than 10 000 elements per PE. Note that certain optimizations in RAxML-NG are disabled, as they lack bit-reproducible implementations (Section 5.4). Nonetheless, based on these measurements, we are cautiously optimistic that a bit-reproducible phylogenetic tree inference is feasible, despite having to execute thousands of all-reduce operations per second. We anticipate that this also applies to a bit-reproducible Bayesian phylogenetic inference; investigating this constitutes further work.

5.6 Conclusion and Future Work

Maximum-Likelihood based phylogenetic tree inference tools, such as RAxML-NG, heuristically search for the phylogenetic tree that best explains the observed genomic input sequences (Sections 2.1 and 2.3). However, due to differences in rounding errors of IEEE 754 floating-point computations during all-reduce operations, parallel phylogenetic tree inferences frequently diverge under varying PE-counts. Supporting Shen *et al.* [She+20a], we observe that the level of parallelism contributes to diverging phylogenetic tree searches on 31 % of 10 130 empirical datasets. Moreover, 8 % of diverging datasets yield trees that are significantly worse than the best found tree (AU-test, $p < 0.05$). To alleviate this issue, we present Repro-RAxML-NG, a variant of the widely-used phylogenetic tree inference tool RAxML-NG that yields bit-reproducible results under varying core-counts. This variant is merely 0 to 6.7 % (median 0.93 %) slower than the non-reproducible reference on up to 768 PEs.

Moreover, we present the bit-reproducible reduction algorithm ReproRed, which maintains the order of reduction operations and is independent of the communication pattern. To the best of our knowledge, ReproRed is thus the first bit-reproducible distributed-memory reduction algorithm that allows for arbitrary associative reduction operators. This permits the integration of ReproRed into the MPI wrapper library KaMPIng, which enables users to pass arbitrary reduction operators, such as C++ lambdas or function pointers. ReproRed only exchanges the theoretical minimum number of messages, overlaps communication with computation, and utilizes a fast base-case implementation for local reductions. As an example, we implement an AVX2-vectorized base-case for summation as the reduction operator.

We find that ReproRed is able to all-reduce (via a subsequent broadcast) 4.1×10^6 elements on 48 to 768 PEs in 19.7 to 48.61 µs. Thus, ReproRed exhibits a slowdown of 13 to 93 % over an non-reproducible all-reduce algorithm and 8 to 25 % over a non-reproducible Reduce-Bcast algorithm. ReproRed outperforms the state-of-the-art reproducible all-reduce algorithm ReproBLAS when more than 10 000 elements reside on each PE.

In conclusion, we highlight the issue of non-reproducibility in phylogenetic inference and present the first bit-reproducible phylogenetic tree inference tool. Additionally, we incorporate our reduction algorithm, which supports arbitrary reduction operators, into an open-source MPI-wrapper library, thereby facilitating the transition to bit-reproducible code.

In future work, we aim to develop bit-reproducible kernels for calculating the derivatives

of the phylogenetic likelihood and to port optimizations like pattern compression [SL03] to Repro-RAxML-NG. We also intend to implement the recursive-doubling all-reduce variant of ReproRed (Section 5.3.2), thus speeding up all-reduction. Additionally, we plan to extend ReproRed to support both, inclusive, and exclusive prefix sums.

6 KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI

Attribution: Large parts of this chapter have been copied either in modified or verbatim form from this the peer-reviewed publication "KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI" [Uhl+24a], where I am a (shared) first author. In the following, I break down my contribution to KaMPIng, a large-scale project with multiple person-years of work by eight distinct authors.
Florian Kurpicz, Tim Niklas Uhl, Lukas Hübner (I), Matthias Schimek, Daniel Seemaier, and Demian Hespe (names in random order) contributed equally to the core functionality of KaMPIng. This includes the design, implementation, and evaluation of (a) named parameters that can be provided in arbitrary order with automatic computation and transfer of omitted information, (b) MPI communicator abstractions as well as initialization and clean-up of the MPI environment, MPI groups, and contiguous blocks of memory, (c) compile-time checks with readable error messages, and (d) design of the plugin infrastructure.
Abstractions of MPI calls are closely linked to KaMPIng's core, including named parameter parsing, safety checks, automatic omitted parameter deduction, and buffer ownership management. These functions were often developed by a single author and later refactored, refined, and extended by others. My major contributions are (a) the initial implementation of most (blocking) collective operations, which was later refactored and refined by Matthias Schimek and Tim Niklas Uhl, (b) implementing automatic compile-time deduction of the MPI type from the provided C++ type, later extended by Tim Niklas Uhl, (c) co-design and implementation of the plugin hooks system in close collaboration with Tim Niklas Uhl. Additionally, I contributed via code reviews or small code additions to numerous other features.
Most of KaMPIng's high-level features were developed by one or two authors each. I implemented the abstraction over MPI's failure-tolerance layer and integrated KaMPIng into RAxML-NG, our largest real-world application benchmark. Further, I supervised and reviewed the inclusion of a reproducible reduction algorithm into KaMPIng (Chapter 5) by Christoph Stelz. My minor contributions include initial versions of sparse collectives and custom data types.
Further, I contributed by (a) creating documentation and examples together with Matthias Schimek, (b) developing coding, documentation, testing, and contribution guidelines together with Florian Kurpicz, (c) performing experiments regarding KaMPIng's integration into RAxML-NG, (d) conducting requirements engineering, later refined by Tim Niklas Uhl.
Note that I omitted those parts of KaMPIng in this breakdown to which I did not substantially contribute. These include our custom assertion library KAssert, point-to-point communication, management of memory allocation and ownership, automatic serialization, and application benchmarks other than RAxML-NG.
Peter Sanders supervised the entire project, was involved in high-level discussions, set goals, and provided guidance.

Summary: The Message Passing Interface (MPI) and C++ form the backbone of High Performance Computing (HPC), with MPI offering abstractions over exchanging messages among processes in distributed-memory HPC systems. However, MPI only provides C and FORTRAN bindings, even though high-level programming languages such as C++ yield software development quicker and less error-prone [BN11].
We therefore propose a novel set of C++ language bindings as well as an extensively tested, open-source implementation (KaMPIng), covering all abstraction levels from low-level MPI calls to convenient STL-style bindings. Through configurable inference of parameter defaults, fine-grained memory allocation control, enhanced safety guarantees, and a flexible plugin system, KaMPIng enables both, rapid prototyping and careful fine-tuning of distributed algorithms. Further, KaMPIng provides this added functionality at (near) zero runtime overhead by leveraging template-metaprogramming, causing the compiler to only generate the necessary code paths.
We demonstrate that KaMPIng provides a foundation for a future distributed parallel programming standard library using multiple application benchmarks, from sorting algorithms to phylogenetic interference using RAxML-NG. Additionally, we integrate a bit-reproducible distributed-memory reduction algorithm as well as abstractions over MPI's failure-mitigation interface into KaMPIng. Various scientific C++ codes already utilize KaMPIng and we are working closely with the MPI language binding working group towards the development of a novel MPI C++ interface with KaMPIng as a prototypical reference implementation. By incorporating further highly-efficient basic toolbox algorithms into KaMPIng, and eventually covering the entire MPI standard, we strive to establish KaMPIng as a stable core for a comprehensive ecosystem of future general-purpose distributed algorithms and applications.

6.1 Introduction

The growing demand for efficiency, parallelism, and bit-reproducibility further increase software complexity (Section 1.1). Abstraction is one of *the* fundamental tools to handle complexity in software systems. Software libraries bundle abstractions and provide sets of related functionality, for example, for exchanging messages in distributed memory High Performance Computing (HPC) systems. Here, the Message Passing Interface (MPI) [MPI4.1] offers abstractions for passing messages between processes in a HPC context (Section 2.6).

The first version of MPI was proposed by the Message Passing Interface Forum in 1994 with the goal of standardizing a portable, flexible, and efficient standard for message-passing [MPI4.1]. Today, it is the message exchange library of choice for most HPC applications and a backbone of scientific computing. While the majority of scientific applications are written in C++ [Lag+19], MPI's syntax and semantics are designed around C and FORTRAN. Although this allows C++ code to invoke MPI's C interface, the semantics do not reflect modern C++ language features such as object orientation, structured bindings, or move semantics (Section 6.2). This yields developing MPI applications in C++ counter-intuitive and error-prone [Rue+21].

Past efforts to introduce C++ bindings for MPI have failed. For instance, the C++ bindings introduced by MPI 2.0 in 1997 were deprecated 12 years later with MPI 2.2. This decision was taken because the bindings only added minimal functionality over the C bindings while inducing substantial maintenance complexity to the MPI specification [MPI2.2]. For example, the official C++ interface did not automatically free resources using the Resource Acquisition Is Initialization pattern [SS+24, P.8] or explicitly transfer ownership of resources

```cpp
std::vector<double> v = {...};
// KaMPIng allows concise code
// with sensible defaults ... (1)
auto v_global = comm.allgatherv(send_buf(v));

// ... or detailed tuning of each parameter (2)
std::vector<int> rc;
auto [v_global, rcounts, rdispls] = comm.allgatherv(
    send_buf(v), //(3)
    recv_counts_out<resize_to_fit/*(6)*/>(std::move(rc)), //(4)
    recv_displs_out() //(5)
);
```

Figure 6.1 KaMPIng offers a high-level easy-to-use interface (1) and full control over each parameter (2). Data types and buffer sizes can be automatically inferred (3). Arguments allow for passing by reference or by value and transferring data ownership via move semantics (4). Out-parameters allow for controlling which parameters will be returned to the user (5). Resize policies allow for controlling memory allocation (6).

using move semantics [SS+24, page I.11]. This was in part due to the respective C++ features – such as move semantics – not being available at the time. Subsequently, MPI's C++ bindings have been removed entirely with version 3.0.

Since then, further C++ interfaces for MPI have been proposed. Notable libraries include *Boost.MPI* [GT07], the MPI bindings by Demiralp *et al.* [Dem+23], and *MPL* [Bau15]. The latter has recently been considered as a starting point for new standardized C++ language bindings by the recently formed MPI working group on language bindings [Gho+21].

Previous bindings focused on yielding MPI's C interface compatible with C++ features such as templates, containers from the C++ Standard Template Library (STL), object orientation, and providing sensible defaults for optional parameters. However, this resulted in each library choosing its own level of abstraction: They either provide a high-level interface, while often sacrificing performance [Gho+21], or only offer a thin layer of abstraction over MPI's C interface, still necessitating substantial amounts of boilerplate code. Additionally, these libraries often do not support explicitly transferring memory ownership via move semantics (Section 6.2), and other common practices proposed by the C++ core guidelines, for instance, returning results by value instead of using C-style in/out parameters [SS+24, F.20]. Therefore, despite past efforts, designing an improved C++ MPI interface remains an open problem that is being actively discussed in the MPI forum [MPI20].

As a solution, we propose a novel set of C++ MPI bindings and our respective open-source implementation *KaMPIng* (Karlsruhe MPI next generation).[1] KaMPIng's main design goal is to cover the complete range of abstraction levels over MPI calls (Figure 6.1). It enables rapid prototyping, relying on its sensible defaults, as well as fine-tuned distributed algorithms by allowing the user to freely chose the degree of automation added on top of the MPI interface. Further, KaMPIng supports inference of defaults for optional parameters, for example, the receive-counts and displacements during an `allgatherv` operation (Section 6.2), thereby substantially reducing boilerplate code. Moreover, KaMPIng offers extensive control over memory allocations, for instance, allowing the caller to re-use buffers or automatically resizing them to the required size (Section 6.4.3). KaMPIng also helps to reduce common sources of programming errors by employing compile-time error checking (Section 6.4.7) and prevents

[1] `https://github.com/kamping-site/kamping`; LGPL

invalid memory access in non-blocking communication via an ownership model (Section 6.4.5). KaMPIng's flexible type system supports type safety by generating type definitions at compile time (Section 6.4.4) and enables serialization when needed (Section 6.4.4). Further, KaMPIng provides a plugin system, which we use, for example, to provide abstractions for MPI's failure-mitigation mechanism (Section 6.6.2) and a bit-reproducible distributed-memory reduction algorithm (Section 6.6.3). KaMPIng provides this added functionality at (near) zero runtime overhead by leveraging template-metaprogramming, causing the compiler to generate only those code paths that are necessary for a specific call site. That is, if a specific call site to KaMPIng does not request automatic computation of, for example, the send-counts, the compiler will not even generate (always false) branch instructions.

Currently, KaMPIng supports the most commonly used MPI features [Lag+19], namely blocking and non-blocking point-to-point communication as well as collective communication (Section 2.6). Moreover, its core architecture is designed with the rest of the MPI standard in mind, facilitating a straight-forward implementation of it in the future.

We highlight KaMPIng's flexibility by discussing various application benchmarks, including simple examples and complex real-world applications. This includes sorting, text processing, graph search, and graph partitioning algorithms as well as a real-world phylogenetic inference tool. For instance, KaMPIng enables a Prefix Doubling for suffix array construction [MM93] implementation using less than 200 lines of code (Section 6.5.1) and to entirely remove the custom MPI abstraction layer from RAxML-NG [Koz+19] (Section 6.5.3).

Contribution

In this chapter, we provide a detailed overview over previous efforts to design C++ MPI bindings. Learning from their efforts and the MPI community, we propose a novel set of C++ bindings for MPI and make available KaMPIng, a respective open-source implementation. It enables rapid prototyping, relying on its sensible defaults, as well as fine-tuned distributed algorithms by allowing the user to freely chose the degree of automation added on top of the MPI interface. KaMPIng's novel approach to parameter handling frees application developers from remembering MPI's convoluted function signatures with up to 10 parameters [MPI4.1, ch. 6.12] and allows them to flexibly decide which parameters to provide and which parameters KaMPIng shall infer. Further, KaMPIng permits highly-engineered distributed applications to exert fine-grained control over memory allocation, in order to, for instance, re-use or automatically resize buffers. KaMPIng also helps to reduce common sources of programming errors by employing compile-time error checking and preventing invalid memory access during non-blocking communication via an ownership model. KaMPIng's type system supports generalization and type safety (Section 6.2) by generating type definitions at compile time and provides serialization when requested. KaMPIng provides this added functionality at (near) zero runtime overhead by leveraging template-metaprogramming, causing the compiler to generate only those code paths that are necessary for a specific call site. Further, we provide high-level basic toolbox algorithms via KaMPIng's plugin system, for example, a bit-reproducible distributed-memory reduction algorithms as well as specialized collectives for sparse and low-latency irregular all-to-all message exchanges. We demonstrate KaMPIng's applicability to real-world problems via a plethora of application benchmarks, including string and graph algorithms as well as integration into the widely-used phylogenetic inference tool RAxML-NG [Koz+19].

Overall, KaMPIng substantially reduces the verbosity and error-proneness of MPI code, at (near) zero runtime cost over plain-MPI. Various scientific C++ codes already use KaMPIng, and it will further facilitate the rapid development of highly-tuned distributed algorithms. We

strive to establish KaMPIng as a stable core for a whole ecosystem of future general-purpose distributed algorithms and applications. For this, we are closely collaborating with the MPI language binding working group towards the development of a novel MPI C++ interface with KaMPIng as a prototypical reference implementation. Further, we are working on incorporating further highly-efficient basic toolbox algorithms into KaMPIng, and eventually covering the entire MPI standard.

Outline of this Chapter

The remainder of this chapter is structured as follows: First, we provide background information on MPI collectives touched upon in this chapter as well as relevant C++'s language features (Section 6.2). Next, we discuss existing MPI (C++) language bindings (Section 6.3) and present KaMPIng's core features such as parameter handling, type deduction and serialization, handling of non-blocking communication, and a plugin-system (Section 6.4). In Section 6.5, we demonstrate KaMPIng's applicability to several real-world applications. Finally, we present first examples for general distributed-computing algorithmic building blocks (Section 6.6) and conclude by discussing avenues of future work (Section 6.7).

6.2 Preliminaries

In this section, we provide background information required for understanding the remainder of this chapter.

Collective Operations Next to point-to-point messages, MPI also supports abstract collective operations (Section 2.6), such as `reduce` and `allreduce` (Chapter 5). In this chapter, we also consider variants of the `gather` and `alltoall` collective operations. Moreover, while the `gather` operation collects data from all involved Processing Elements (PEs) and returns a concatenation on a single *root* PE [MPI4.1, ch. 6.5], `allgather` returns the concatenation of the collected data on all PEs [MPI4.1, ch. 6.7]. The `alltoall` operation abstracts a message exchange between all pairs of involved PEs [MPI4.1, ch. 6.8].

Additionally, the `allgather` and `alltoall` collectives provide variants which allow the amount of data sent by PEs to vary. MPI denotes these operations as `allgatherv` [MPI4.1, ch. 6.5] and `alltoallv` [MPI4.1, ch. 6.8], respectively. `Alltoallw` further generalizes the message exchange, allowing each PE to also specify different data types [MPI4.1, ch. 6.8].

Sparse All-to-all The all-to-all communication pattern requires $\Theta(p^2)$ messages to be exchanged over the network, where p is the number of PEs. As this does not scale well, algorithms are often designed to avoid such a full all-to-all message pattern, exchanging only a subset of these messages in each communication round. For example, a common pattern in HPC is to partition the problem at hand into separate subproblems, which are subsequently assigned to separate PEs. That is, we distribute the *work* and *data* associated with this work across the PEs. However, a single data point might be accessed by multiple work packages, and thus, each PEs shares a subset of its data with a small set of other PEs; we denote this shared data a *halo*. In a *halo exchange*, PEs exchange the data located in those halos. MPI provides support for such communication patterns, called *neighborhood collective communication* [MPI4.1, ch. 8.6]. For this, the user can, for example, arrange the PEs in a grid of arbitrary dimension [MPI4.1, ch. 8.5.1], allowing each PE to communicate with neighboring PE in the respective virtual topology [MPI4.1, ch. 8.5].

While halo exchanges are useful for regular communication patters often found in numeric applications, MPI also offers abstractions for arbitrary sparse all-to-all communication – that is, where $\mathcal{O}(p)$ out of the p^2 messages are non-empty (Section 6.6.1). MPI offers such message exchanges via the construction of arbitrary virtual topologies – representing arbitrary communication patters [MPI4.1, ch. 8.5]. However, creating these virtual topologies incurs a runtime overhead and is thus less suitable for dynamic communication patterns. Further, MPI requires each PE to specify each recipient of an outgoing message, and sender of an incoming message during creation of a virtual topology [MPI4.1, ch. 8.5.4]. While this allows MPI to perform pairwise exchanges, which are faster than separate send and receive operations, it also requires each PE to know from where it will receive messages. In Section 6.6.1, we add further algorithms for sparse all-to-all communication, which do not require setting up a virtual topology – thus incurring no runtime overhead – and no knowledge of where a PE will receive messages from.

In-Place Message Exchanges MPI offers *in-place* message exchanges, which do not require distinct send and receive buffers, but overwrite the send buffer contents with the received data. For this, the user has to pass the special constant `MPI_IN_PLACE` as the *send* buffer, which causes MPI to send the data from and receive data the "receive" buffer. For in-place operations, the send count, send displacement, and send type parameters are ignored. In contrast, KaMPIng requires only a single `send_recv_buf` parameter for in-place calls (Section 6.4.7).

```
// Gather a single element from each PE; return the concatenation of these
// elements on all PEs. receive_buffer must be at least numbers of PE large.
MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
    receive_buffer, 1, MPI_INT, comm);
```

Polymorphism and Type Safety In order to facilitate code re-use, functions in C++ (and many other languages) can be parameterized to operate on different data types [Str00]. We denote this as *polymorphism*. In MPI, many parameters to functions are provided as `void*`, that is, a pointer to a type-less memory location. Consider, for instance, the send and receive buffers in the MPI `allreduce` call [MPI4.1, ch. 6.9], which are both of type `void*`, with the data type provided as a special constant in a separate parameter. Thus, the same function is called, for instance, for integer and floating-point values. However, this interface design prevents the compiler from enforcing, or checking, that the actual data type matches the declared data type. Further, the compiler can not check if the same data type is provided to the same collective operation on all PEs.

In contrast to C, C++ offers so-called parametric polymorphism [Str00] in the form of templated functions (and classes). This allowing for the compiler to prohibit type errors, a concept we denote as *type safety* [Mil78] in this chapter.

Move Semantics C++ 11 introduced move semantics, which allow to semantically transfer object ownership when invoking a function – including the responsibility to free the underlying memory. In contrast, passing a parameter by value creates a copy – incurring a runtime cost – and passing it by pointer or reference creates an alias – leaving ownership management to the user. We employ move semantics in order to transfer buffer ownership between the caller and KaMPIng (Section 6.4.3). This simplifies memory management and enables us to return out-parameters by value in accordance to the C++ core guidelines [SS+24, F.20]

while avoiding an expensive deep copy (Section 6.4.2). Further, it nudges the user away from invalid access to buffers used in asynchronous communication (Section 6.4.5).

Structured Bindings C++'s *structured bindings* provide a shorthand for binding the elements of a structured object to multiple variables [SB25]. This allows for conveniently returning multiple variables from a single function invocation:

```
// f() returns a std::tuple with tree elements, which are bound to the
// names a, b, and c.
auto [a, b, c] = f();
```

6.3 Related Work

Since the deprecation of the official C++ bindings from the MPI standard, there have been continuous third-party efforts to design novel, improved C++ language bindings. The topic is also actively discussed in the MPI forum [MPI20]. For example, a C++ binding should automatically infer the MPI data types from the C++ data types for both, basic data types such as `int` as well as complex structs. Further, the bindings should increase type safety over MPI's approach of employing `void` pointers for every buffer. Moreover, C++ bindings should enforce the MPI requirement of callers not accessing send and receive buffers during non-blocking communication. Next, the C++ core guidelines strongly suggest that functions return output data by value [SS+24, F.20], which requires an MPI wrapper library to abstract over MPI's C-style input and output parameters. Further features requested in the MPI forum [MPI20] are a strong debug mode, including extensive error checking, the ability to provide reduction operators via C++ lambdas, as well as interaction with modern C++ features such as move semantics and ranges.

Boost.MPI [GT07] Boost.MPI was the first library that was capable of automatically inferring MPI data types. This facilitates integration with the STL by supporting containers such as `std::vector` for input and output next to raw pointers. Boost.MPI automatically resizes vectors to fit the received data, which prevents invalid memory accesses, but induces hidden memory allocations and copy operations. Further, Boost.MPI supports data serialization next to the construction of MPI data types for transferring non-trivial C++ types. In order to enable serialization, users must provide a serialization function that is compatible with *Boost.Serialization*, specifying all class members explicitly. This requires the user to explicitly synchronize the data type definitions and their serialization functions. Boost.MPI manages custom data types via a global type pool, which it has to query before each message exchange in order to ensure that resources are properly cleaned up, thereby incurring a runtime overhead.

The adoption of Boost.MPI has been hindered by its tight coupling with other Boost libraries and by its implicit serialization – and the associated performance pitfall – of data types that are not directly supported by MPI [Gho+21]. Additionally, Boost.MPI is the only MPI library considered here that is not header-only. As the MPI standard does not guarantee that code compiled against one MPI implementation will execute correctly with any other MPI implementation,[2] a separate build of Boost.MPI is required for each installed MPI library. Further, besides KaMPIng, Boost.MPI is the only library that supports mapping

[2] This will change in the upcoming MPI 5.0 standard. [MPI5.0, ch. 20]

STL functors such as `std::plus` to the corresponding built-in MPI constant for reduction operations (which allows optimization by the MPI implementation) and writing custom reduction operations via C++ lambdas. If an MPI error occurs, Boost.MPI throws an exception. Boost.MPI does not provide any bindings for `MPI_Alltoallv` and has not been actively maintained since 2008 and therefore only supports MPI-1.1 features.

However, *boost-mpi3* [Cor18] is a rewrite of Boost.MPI, developed independently of the Boost project but following its design principles. The project aims to extend Boost.MPI to the feature set of MPI-3 and C++ 17. While boost-mpi3 includes support for iterators, MPI one-sided communication, and MPI shared memory, it does not substantially improve upon Boost.MPI's design, still closely mirroring MPI's C interface.

Message Passing Library (MPL) [Bau15] Bauke developed *MPL*, a library providing MPI bindings targeting C++17. MPL introduces a type system that allows user to programmatically construct views over contiguous memory chunks, which MPL subsequently converts into MPI data types. This allows to create and exchange strided data types, for example, a column in a matrix which is stored row-first in memory. Moreover, this allows the user to exchange containers of dynamic size, for example, a `std::vector`; however, MPL creates a new type each time the container size changes. MPL does not expose the underlying native MPI representation of communicators and data types, which complicates an iterative adaptation of existing MPI code to MPL. MPL offers support for custom data types, which require defining a matching layout but does not offer serialization support. Further, MPL currently does not support error handling.

Recently, MPL has been considered as a starting point for new C++ bindings by the members of the newly formed MPI working group on language bindings [Gho+21]. While Gosh *et al.* [Gho+21] highlight the simplified interface, which offers abstraction with default parameters using function overloads, they also show that MPL incurs runtime overheads for variable-size collectives. This is because MPL does not perform the communication directly via the corresponding collective operations using appropriate counts and displacements, but rather constructs custom derived data types with memory offsets. Therefore, abstractions over collectives such as `gatherv` (Section 6.2) internally call `MPI_Alltoallw` (Section 6.2), which is costly in some MPI implementations, thereby limiting scalability [Gho+21].

RWTH-MPI [Dem+23] Demiralp *et al.* recently introduced a modern C++ interface, which we denote RWTH-MPI (Rheinisch-Westfälische Technische Hochschule (Aachen) MPI) in the following. RWTH-MPI offers full support for sending and receiving STL containers. For each communication call, RWTH-MPI provides various function overloads using different abstraction levels, which often allow omitting send or receive-counts, causing RWTH-MPI to automatically compute them, inducing (necessary) additional communication. Automatic resizing of the receive buffer is supported in some cases but can be disabled. For custom types that support reflection based on the PFR library [Pol16], RWTH-MPI automatically constructs appropriate MPI data types. A customization point for mapping C++ types to MPI data types is available, but does not provide automatic de-allocation of MPI data types. Using types with dynamic run-time sizes is not supported.

While RWTH-MPI covers the complete MPI standard, large parts directly mirror the C interface without providing additional abstractions, convenience, or safety guarantees.

Beyond C++ MPI bindings for high-level programming languages beside C++ exist. The Python bindings *mpi4py* [Dal+08; DF21] have been actively developed for over a decade

and may therefore be considered as the most mature third-party bindings. They enable the transparent sending of Python objects using serialization. While this incurs additional runtime overhead, using mpi4py in conjunction with NumPy arrays yields performance that is on par with native C implementations [Dal+08]. Further, they provide default values for some parameters, but only if no additional communication is necessary to infer them. Finally, rsmpi [SGB15] introduces idiomatic Rust bindings for MPI. While rsmpi makes calling MPI from Rust more ergonomic than via the C interface, the missing default parameters require writing a considerable amount of boilerplate code.

6.4 KaMPIng Design

Realizing a C++ interface that is both, easy to use, and highly flexible, requires building a library from scratch and improving upon the design of its precursors. Similar to most previous bindings, KaMPIng represents MPI objects such as communicators, requests, and statuses as classes, and operations on them as member functions. We implement automatic creation and destruction of MPI objects via the *Resource Acquisition Is Initialization* [Str94] design pattern, as recommended by the C++ core guidelines [SS+24, E.6]. Further, we map C++ data types to MPI types at compile time, thereby preventing type-matching errors. KaMPIng directly supports STL containers that allow access to the underlying contiguous memory, that is, every container that models the `std::contiguous_range` concept. Further, KaMPIng supports raw pointers via `std::span` as proposed in the C++ core guidelines [SS+24, page I.13].

Named Parameters A common source of programming errors in MPI stems from the interface complexity of many communication calls. For example, the group of *variable* collective operations (suffixed with `v`, see Section 6.2) allows the amount of data transferred between each PE-pair to vary [MPI4.1, ch. 6]. Thus, variable collectives require a large number of function parameters, for example, eight for `MPI_Gatherv`. This makes MPI calls verbose and requires programmers (at least the authors) to frequently consult the documentation for writing and reading respective call sites. While all parameters are necessary to provide full flexibility, many use cases exist in which only a small subset of explicitly provided parameters is required, and the remaining parameters can be inferred automatically. KaMPIng introduces named parameters combined with compile-time generation of only the required code-paths, thus freeing users from remembering complex function signatures while still providing the flexibility of optional parameters at zero runtime overhead. To the best of our knowledge, this approach is unique among C++ MPI wrappers. We briefly discuss KaMPIng's parameter handling in Section 6.4.1 and explain in more detail in Section 6.4.2.

Memory Management Memory management is challenging. C++ programmers have to constantly keep track of which part of the code is responsible for de-allocating memory blocks, while avoiding unnecessary memory allocations and copy operations, which incur a runtime cost. Thus, we design KaMPIng such that users can flexibly control the degree of automation KaMPIng exhibits when allocating memory (Section 6.4.3) at compile-time. For example, users can re-use allocated memory or let KaMPIng automatically re-size buffers in order to fit all received or computed data. Further, KaMPIng can be instructed to allocate the required buffers itself, semantically transferring the ownership back to the caller if so instructed (Sections 6.2 and 6.4.3).

Type Inference The missing type introspection features of C require MPI users to explicitly specify the layout of data types they intend to transfer over the network. However, if the type declaration and actual data layout are not explicitly synchronized, this may induce hard-to-find errors. In Section 6.4.4 we introduce KaMPIng's flexible type system which provides type-safety through compile-time construction of MPI data types from C++ data types. Additionally, KaMPIng offers support for runtime-sized types and explicit serialization and deserialization that is transparent to the user (Section 6.4.4).

Safety Guarantees for Non-Blocking Communication Non-blocking communication in MPI introduces additional sources of programming errors, as the caller has to explicitly wait for the completion, avoiding invalid memory accesses to buffers that are associated with a non-blocking operation before it has been completed. To address this issue, we propose memory-safe abstractions that prevent illegal memory accesses via C++'s ownership model and move semantics in Section 6.4.5.

Extensibility via Plugins The MPI standard is continuously growing, requiring C++ bindings to evolve while maintaining compatibility with existing code as well as (new) MPI features not yet covered by such bindings. We thus decided to keep KaMPIng's core small, but allow for easy integration of additional features via a plugin system (Section 6.4.6).

Preventing Common Programming Errors Further, KaMPIng prevents common programming errors by providing a plethora of compile- and runtime assertions, configurable error handling, and introducing a simplified syntax for in-place operations (Sections 6.2 and 6.4.7).

6.4.1 Inferring Default Parameters

Common usage scenarios allow for inferring a large fraction of the parameters passed to an MPI call (Section 6.4). Consider, for instance, `MPI_Allgatherv` (Section 6.2), which collects data from all PEs and returns a concatenation of this data on all PEs. In case the data to be sent are stored in a container (e.g., `std::vector`), the number of elements to be sent (*send count*) as well as the underlying data type can be automatically inferred from the container's size and `value_type` respectively (Section 6.4.4). Further, we can automatically compute the number of elements to be received (*receive count*) and the offset of each remote processes' data in the local receive buffer (*receive displacements*). For this, KaMPIng issues an `MPI_Allgather` in order to collect all send-counts on all PEs. KaMPIng subsequently computes the exclusive prefix sum over them (Figure 6.2) to obtain the receive displacements.

While this is a common pattern in scientific codes, to the best of our knowledge, none of the existing C++ bindings includes a respective automatism to yield boilerplate code obsolete. For example, Boost.MPI offers function overloads that allow the user to omit explicit receive displacements, however, the user still has to explicitly communicate the send/receive-counts. Further, RWTH-MPI provides a function overload that gathers the counts internally, but only works with `MPI_IN_PLACE`, which requires the data to be sent to already be located at the correct memory location on each PE. In order to place the outgoing data accordingly, the user has to manually exchange count information upfront.

In KaMPIng, we chose *named parameters* to allow the caller to provide any subset of possible parameters in an arbitrary order (as familiar from languages such as Python). Internally, we implement these named parameters via *factory functions* [Gam+95] which construct lightweight parameter objects encapsulating the *parameter type* (i.e., send buffer, send-counts,

...) as well as the corresponding data. This allows us to check if the caller provided a specific parameter and to compute default values otherwise – while avoiding the combinatorial explosion that a design employing function overloads would entail. We implement named parameters with zero run-time overhead by using template-metaprogramming to check for a parameter's presence and only generate the code-paths that are required for the parameters given at each specific call site.

Using named parameters, gathering a vector in KaMPIng thereby becomes a one-liner (Figure 6.1), while implementations using other bindings are more verbose (Table 6.1). Moreover, this flexibility provided by named parameters permits users to iteratively adapt their existing code to KaMPIng (Figure 6.3), thereby considerably lowering the entry barrier.

6.4.2 Input and Output Parameters

KaMPIng extends MPI's definition of *in(put)*- and *out(put)*-parameters. Users can provide data to library functions via in-parameters, for example, `send_buf(data)` informs KaMPIng where to find the data to be sent. In contrast, out-parameters, for instance, `recv_counts_out()`, requests the library to *return* the respective data. Further, KaMPIng can automatically deduce most of the in-parameters plain-MPI requires the user to explicitly provide. For example, during an `alltoallv` operation (Section 6.2), KaMPIng can determine the send displacements if the number of elements to be sent to each PE is provided. Note that this sometimes also involves additional communication, for example, in the case of receive-counts (Section 6.4.1). KaMPIng allows the user to pass these (optional) parameters either as in- or out-parameters, the latter implying that KaMPIng will compute them. For example, `send_displs(data)` creates an in-parameter containing the send displacements as specified in the `data` container, whereas `send_displs_out()` creates an out-parameter signaling KaMPIng to compute the send displacements and return them. Moreover, KaMPIng

```
std::vector<T> v = ...; // fill with data

// plain-MPI
int size, rank;
MPI_Comm_size(comm, &size);
MPI_Comm_rank(comm, &rank);
std::vector<int> rc(size), rd(size); // receive-counts and displacements
rc[rank] = v.size();
// exchange the send-counts
MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, rc.data(), 1,
    MPI_INT, comm);
// Compute receive displacements
std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
int n_glob = rc.back() + rd.back();
// allocate receive buffer
std::vector<T> v_glob(n_glob);
// exchange data
MPI_Allgatherv(v.data(), v.size(), MPI_TYPE, v_glob.data(),
    rc.data(), rd.data(), MPI_TYPE, comm);

// KaMPIng
std::vector<T> v_glob = comm.allgatherv(send_buf(v));
```

Figure 6.2 Using plain-MPI to collect data from all PEs, returning the concatenation of the collected data on all PEs (`allgather`). The caller has to write boilerplate code to exchange the send/receive-counts and compute the receive displacements manually. In contrast, KaMPIng will exchange and compute the counts and displacements automatically, resulting in a single line of code.

```
// Version 1: using KaMPIng's interface
std::vector<int> rc(comm.size()), rd(comm.size());
rc[comm.rank()] = v.size();
comm.allgather(send_recv_buf(rc));
std::exclusive_scan(rc.begin(), rc.end(), rd.begin(), 0);
std::vector<T> v_glob(rc.back() + rd.back());
comm.allgatherv(send_buf(v), recv_buf(v_glob),
    recv_counts(rc), recv_displs(rd));

// Version 2: displacements are computed implicitly
std::vector<int> rc(comm.size());
rc[comm.rank()] = v.size();
comm.allgather(send_recv_buf(rc));
std::vector<T> v_glob;
comm.allgatherv(send_buf(v), recv_buf<resize_to_fit>(v_glob),
    recv_counts(rc));

// Version 3: counts are automatically exchanged
// and result is returned by value
std::vector<T> v_glob = comm.allgatherv(send_buf(v));
```

Figure 6.3 Existing plain-MPI code (Figure 6.2) can be incrementally ported to KaMPIng.

permits the user to instruct KaMPIng to re-use already allocated memory for storing the data of an out-parameter (Section 6.4.3). This allows, for instance, to re-use buffers allocated once in frequent communication patterns.

As the receive buffer is of interest at most MPI call sites, KaMPIng always implicitly returns it. The caller has to request all further out-parameters (e.g., receive displacements) explicitly via the corresponding out-parameter.

Existing MPI wrapper libraries (Section 6.3) mimic MPI's C interface and return output data by pointer or by reference. However, this interface design contradicts the C++ core guidelines, which strongly suggest returning output data by value [SS+24, F.20]. Further, when reading the code at a call site, it is often unclear, which parameters are in- and which are out-parameters. This is further complicated by the plethora of function overloads used by existing MPI wrappers to provide the flexibility of optional parameters. In contrast, KaMPIng call sites clearly document which parameters are in- and which out-parameters. Consider, for instance, the following call to `allgatherv` (Section 6.2):

```
auto result = comm.allgatherv(send_buf(v), recv_counts_out());
auto recv_buf = result.extract_recv_buf();
auto counts = result.extract_recv_counts();
```

Here, `allgatherv` returns a result object containing the implicitly requested receive buffer and the explicitly requested receive-counts. The caller can then extract both from the result object using C++'s move semantics (Section 6.2), thus obtaining them by value and taking ownership. This includes the responsibility for freeing their respective memory, which is automatically provided by the respective destructors. Moreover, the result object is decomposable via C++'s structured bindings (Section 6.2), which simplifies the call to:

```
auto [recv_buf, counts] = comm.allgatherv(send_buf(v), recv_counts_out());
```

If the caller has already allocated memory for certain buffers, KaMPIng offers two mechanisms for re-using it: C++'s move semantics or pass-by-reference. The ownership of user-allocated containers that support C++ move semantics can be transferred to KaMPIng by moving the data into the respective call. After performing the communication, KaMPIng moves the respective buffer's ownership back to the result object and thus, the caller.

```
std::vector<T> tmp = ...;
// Ownership of tmp is moved to .allgatherv(), allowing the buffer to be
// re-used. The container is subsequently moved into the recv_buffer.
auto recv_buffer = comm.allgatherv(send_buf(v), recv_buf(std::move(tmp)));
```

In case a user-allocated container does not efficiently support move semantics, KaMPIng also allows the caller to pass a buffer by reference. Subsequently, KaMPIng writes the data computed/transferred for the requested parameter directly to the specified memory location.

```
std::vector<T> recv_buffer = ...;
// The received data is directly written into recv_buffer.
comm.allgatherv(send_buf(v), recv_buf(recv_buffer));
```

6.4.3 Controlling Memory Allocation

Existing MPI wrappers (Section 6.3) have no unified way of controlling memory allocation. They either always resize user-provided containers to fit the received data, or force the user to pass raw pointers in order to prevent resizing. Further, they do not provide mechanisms to re-use buffers for automatically computed parameters. In contrast, KaMPIng allows the caller to exert fine-grained control over memory management. Each (out-)parameter that accepts a container, takes an optional template parameter, which indicates its *resize policy*. This resize policy controls, whether the respective container is always resized to fit, resized if it is too small to store the result, or that KaMPIng shall perform no checks and assume that the capacity of the container suffices. As the latter does not induce any runtime overhead, it is the default setting.

```
std::vector<T> recv_buffer; // too small, has to be resized
std::vector<int> counts(comm.size()); // sufficiently large
comm.allgatherv(send_buf(v), recv_buf<resize_to_fit>(recv_buffer),
    recv_counts_out(counts));
```

In case KaMPIng has to create auxiliary data structures to compute missing parameters, the caller may either specify a pre-allocated container or provide the type to be allocated. This allows, for example, to instruct KaMPIng to create a container using a memory allocator optimized for multi-thread applications. Recall, that additional allocation is entirely omitted when parameters are provided by the user.

We implement all of this via template-metaprogramming. Thus, there is no additional overhead compared to a implementation using plain-MPI. KaMPIng's design of optional parameters allows for rapid prototyping and full-grained control over memory allocation, for example to avoid superfluous re-allocation of memory.

6.4.4 Custom Data Types

HPC applications communicate a plethora of data types, ranging from basic data types such as `int` or `float` – which directly map to MPI types – to complex structs with alignment gaps and strided data types (Section 6.3). MPI allows for communicating such complex *derived data types* using type constructors, such as `MPI_Type_create_struct` and `MPI_Type_create_contiguous`. However, C's lack of type introspection forces users to explicitly provide type information, which is both, tedious and error-prone, as type definitions need to be explicitly synchronized with the actual data layout.

C++ types are known at compile-time and can thus be mapped one-to-one to MPI types during compilation using template-metaprogramming. We denote such types as *static*

```cpp
struct MyType {
   int a;
   double b;
   char c;
   std::array<int, 3> d;
};

// KaMPIng's built-in struct serializer can be used to construct a
// MPI_Datatype matching a given struct.
template <>
struct mpi_type_traits<MyType> : struct_type<MyType> {};

// The user can also explicitly construct a MPI type.
template <>
struct mpi_type_traits<MyType> {
   static constexpr bool has_to_be_committed = true;
   static MPI_Datatype data_type() {
       MPI_Datatype type;
       MPI_Type_create_*(..., &type);
       return type;
   }
};
```

Figure 6.4 Defining custom static types using automatic type reflection or a custom type definition.

types. However, MPI's derived data types form a superset of C++ data types, as MPI is able to construct arbitrary type signatures with sizes that are known at run-time only. Concretely, MPI defines a derived data type as an arbitrary number of basic data types with at arbitrary displacements [MPI4.1, ch. 5.1]. Further, these displacements are neither required to be positive, distinct, or provided in increasing order. We denote these as *dynamic types*. KaMPIng supports static and dynamic types and is able to construct static MPI types from C++ types transparently to the user, with predictably low runtime overhead.[3] Sometimes, applications need to communicate unstructured and complex data types off the critical code path. In order to support this with minimal code overhead, KaMPIng supports transparent data serialization, which we discuss in Section 6.4.4.

Deriving Static Data Types at Compile Time KaMPIng maps basic C++ data types to their MPI counterparts and supports more complex data types on homogeneous systems if they are *trivially copyable*, that is, the C++ standard guarantees that they can be copied into a `char` array. For this, KaMPIng creates a contiguous type using `MPI_Type_contiguous` with the appropriate number of bytes. We decided on using `MPI_Type_contiguous` over `MPI_Type_struct` as it exhibited a lower performance overhead in preliminary experiments – as predicted in the MPI standard [MPI4.1, ch. 5.1]. For types that are not trivially copyable, the user can describe how to construct a matching `MPI_Datatype` via an explicit instantiation of the `mpi_type_traits` template for the desired type. However, maintaining code that ensures the correct construction of a MPI type for a given C++ struct is error-prone, as the programmer has to keep the type-construction calls in sync with the data type. Thus, we provide integration with the *PFR* type reflection library [Pol16] in order to automatically generate MPI type definitions for user-provided structs at compile-time (Figure 6.4).

[3] The runtime overhead is unavoidable, as MPI allows creating types only at run-time.

MPI requires the user to initialize and de-allocate non-built-in types. KaMPIng implements both transparently to the user via the Resource Acquisition Is Initialization [Als+04] and the Construct-On-First-Use[4] idioms. Boost.MPI manages types via a global type pool. However, each operation incurs a type lookup at runtime. MPL and RWTH-MPI use the Construct-On-First-Use idiom to commit types before first use, and MPL also provides automatic type definitions via reflection. However, as opposed to KaMPIng, types are not properly freed, which may result in resource leakage. RWTH-MPI allows for custom static type definitions, but the user is responsible for committing and freeing types.

Dynamic Types As mentioned above, dynamic types are created at run-time, not at compile-time. For this, MPL provides a run-time type interface, which mirrors MPI's type constructors via the builder pattern. In contrast, RWTH-MPI does not support dynamic types. Currently, KaMPIng does not offer an abstraction over run-time creation of types. However, it supports passing arbitrary, dynamically created MPI types – that were created via MPI's respective functionality – to KaMPIng functions via optional type parameters. While MPL's type construction system results in verbose code for many use cases, it also constitutes a powerful feature, which we plan to integrate into KaMPIng in future work.

Communicating arbitrary data using serialization Some applications require communicating non-contiguous data which is (partially) located on the heap, for instance, `std::string` or `std::unordered_map`, and can thus not be represented using MPI data types. KaMPIng facilitates the necessary re-arrangement of this data into a contiguous buffer before communication by providing a tuneable serialization functionality that is transparent to the user (Figure 6.5). For this, we employ the C++ serialization library *Cereal* [GV17], which supports STL data types and provides serialization routines for custom types. While users never see serialized data, they have to explicitly enable serialization, as it incurs a hidden cost for allocating memory and performing (de-)serialization. Through Cereal's flexible design, serialization in KaMPIng is also highly configurable; allowing users to specify custom serialization functions and archives, for instance, binary formats, JSON, or XML.

```
using dict = std::unordered_map<std::string, std::string>;
dict data = ...;
comm.send(send_buf(as_serialized(data)), ...);

dict recv_dict = comm.recv(recv_buf(as_deserializable<dict>()));
```

Figure 6.5 KaMPIng can transparently serialize and deserialize data. As this incurs overhead for allocating memory and performing (de-)serialization, the user has to explicitly enable this feature.

Boost.MPI is the only MPI wrapper library besides KaMPIng that offers data serialization. However, Boost.MPI tightly couples serialization with other Boost libraries and implicitly performs serialization if a type is not marked as an MPI data type. This design yields the inherent cost of serialization opaque to the user. As we designed KaMPIng as a zero-overhead library, we chose to require the user to explicitly enable serialization, facilitating a conscious choice regarding the associated runtime and memory overhead.

[4] https://isocpp.org/wiki/faq/ctors#static-init-order-on-first-use

Type Creation for Types with Alignment Gaps For structs with alignment gaps between members, MPI does not include the gap in the communicated data. However, this implies non-contiguous memory accesses, which may be slower than copying entire memory blocks to the communication hardware [MPI4.1, ch. 5.1.6]. As a solution, the MPI standard suggests introducing dummy `struct` members to fill alignment gaps. However, this requires the user to modify their non-MPI code. In contrast, KaMPIng maps trivially copyable C++ types to a MPI type interpreted as a contiguous sequence of bytes, thereby including the alignment gaps. This enables contiguous copying of the data to the communication hardware at the cost of exchanging additional data. We choose this design as preliminary experiments showed this to be faster in practice (data not shown).

6.4.5 Safety for Non-blocking Communication

Non-blocking communication in MPI allows to overlap communication with computation. In MPI, the user can *initiate* a communication operation, obtaining a *request handle*. The user subsequently has to *complete* the request, by either (non-blockingly) testing for completion via `MPI_Test` or by calling `MPI_Wait`, which blocks until the request is completed. MPI's semantics prohibit writing to the send buffers and reading from the receive buffers associated with this communication operation while it is initiated but not yet completed. However, MPI can not actively hinder users from accessing the respective memory locations regardless of completion status, thus introducing an additional source of programming errors.

The C++ STL provides `std::future` for such asynchronous (I/O) operations, which allows for querying or waiting for the result of an asynchronous operation. It only returns a value once the operation has been completed. However, `std::future` cannot be used to provide a safe interface for non-blocking communication, as it assumes that asynchronous progress is made in the background, which the MPI standard does not guarantee for.

Thus, we introduce a concept similar to `std::future`, denoted *non-blocking MPI result*, which encapsulates an `MPI_Request` and the data returned by value from the operation (Section 6.4.2). This ensures, that `result.wait()` returns the data to the user only after completing the request – thereby yielding access to the data valid again. Analogously, calling `result.test()` returns an `std::optional`, which only contains the returned data if the request has been completed, and `std::nullopt`, otherwise. This further prevents semantically invalid access to send buffers during non-blocking operations as the user can move the respective buffer into the communication call, thereby explicitly yielding ownership. KaMPIng subsequently ensures that the respective memory is not freed and returns it to the user upon completion. Thereby, we enforce MPI's semantics for buffer validity in non-blocking communication in C++ (Figure 6.6). Note that this happens without any data copying by solely relying on C++'s move semantics (Section 6.2).

To the best of our knowledge, KaMPIng is the first C++ MPI wrapper library which provides such safety guarantees for non-blocking communication. In contrast, other MPI bindings do not offer enhanced safety features for the data involved in non-blocking calls, but only return request handles – thereby the responsibility to avoid invalid data accesses while the operation has not been completed remains with the user. Only the *rsmpi* Rust library provides analogous guarantees, which are implemented via Rust's ownership model.

Further, KaMPIng provides *request pools*, which allow for convenient management of multiple requests. For this, the user first needs to submit all requests associated with a call to a request pool. The user can subsequently test for the completion of any or all requests.

```cpp
std::vector<int> v = ...;
// Perform a non-blocking send operation. This will return immediately.
// The caller transfers v's ownership to KaMPIng.
auto r1 = comm.isend(send_buf_out(std::move(v)), destination(1));

// Once the send operation completed, KaMPIng moves v's ownership back
// to the caller.
v = r1.wait();

// Perform a non-blocking receive operation.
auto r2 = comm.irecv<int>(recv_count(42));
// ... perform other computations ...
// The data is only returned once the receive finished.
std::vector<int> data = r2.wait();
```

Figure 6.6 Example of non-blocking safety in KaMPIng.

6.4.6 Extensibility

While the main goal of KaMPIng is to design C++ bindings for MPI that are usable in any MPI application, designing a one-size-fits-all library is impossible. We thus designed KaMPIng to permit developers to gradually move their existing code base to KaMPIng. This is possible, because KaMPIng is fully compatible with native MPI objects, such as request or type handles, allowing users to pass them to and extract them from KaMPIng. Further, KaMPIng's plugin interface allows for overriding communicator object member functions (e.g., collectives) and adding additional functionality without changing existing application code. For example, this allows for replacing the MPI implementations pipelining broadcast algorithm with the more efficient 2-3-broadcast [SST09], which is currently not part of any major MPI implementation. Moreover, KaMPIng allows plugins to define new named-parameters to enable the full named parameter flexibility for these library extensions. We chose this plugin architecture in order to keep KaMPIng's core library small and maintainable, while still offering third-party general-purpose functionality. In Section 6.6, we demonstrate the power of KaMPIng's plugin interface by developing and making available multiple plugins that extend the functionality of the current MPI standard.

6.4.7 Additional Safety Features for MPI

KaMPIng offers additional features that aim to prevent common errors when developing MPI programs. Examples include error handling via C++ exceptions, additional compile time checks, and a simplified in-place operation syntax (Section 6.2).

Error handling MPI notifies users of errors by returning an error code. However, MPI does not distinguish between recoverable errors, such as insufficient buffer space, irrecoverable errors, such as hardware failures, and usage errors, such as providing invalid parameters.

KaMPIng improves upon this via three major techniques, following the C++ core guidelines: Throwing exceptions for recoverable errors [SS+24, E.2], catching usage errors at compile time whenever possible [SS+24, P.5], and making heavy use of run-time assertions. While C++'s template-metaprogramming is notorious for complex and hard-to-read compiler errors, we employ `static_assert` to ensure that compile-time assertions fail early and provide helpful human-readable error messages, for instance, for missing parameters or incompatible types. For example, when the user does not provide a required parameter to a collective operation, the compile-time error message indicates which parameter is missing. KaMPIng

further verifies a plethora of MPI invariants during run-time. We group assertions by cost of checking, ranging from lightweight checks to substantial additional communication. Thus, the user can decide which assertions to enable based on the expected overhead. Further, the user can fully disable exceptions and override the error handling strategy using KaMPIng's plugin system (Section 6.4.6). For example, our abstraction over MPI's failure-mitigation mechanisms (Sections 2.9 and 6.6.2) overrides the error handling strategy in order to throw appropriate exceptions when detecting a hardware failure. In contrast, other MPI bindings either always convert MPI errors to exceptions or do not provide any error handling at all.

Simplified `MPI_IN_PLACE` KaMPIng prevents programming errors in the context of in-place MPI operations. When performing an in-place communication in plain-MPI, the user has to explicitly pass `MPI_IN_PLACE` on all PEs as send buffer, causing MPI to ignore the send count and type parameters. KaMPIng's in-out parameters simplify the call semantics of in-place calls by allowing the user to pass data in a `send_recv_buf` instead of a `send_buf`, thus reducing four function parameters to a single one. This causes KaMPIng to automatically pass the correct arguments to the underlying in-place MPI call. Further, this prevents the user from passing `MPI_IN_PLACE` as the receive buffer and issues a compilation error if the user does provide an argument which would be ignored by the in-place call. Further, this is compatible with move semantics, allowing for concise call sites.

```
std::vector<int> data(comm.size());
data[comm.rank()] = ...;
data = comm.allgather(send_recv_buf(std::move(data)));
```

6.4.8 Implementation Details

KaMPIng makes extensive use of C++'s template-metaprogramming capabilities. For example, we often eliminate run-time control flow constructs – preventing costly branch mispredictions – via `constexpr if` and *Substitution Failure is Not an Error (SFINAE)* [VJG18, ch. 8.4]. Further, we employ template parameter packs in order to permit the user to pass named parameters in an arbitrary order. This also allows us to perform compile-time checks for the absence of each parameter, and facilitates us to conditionally enable the compilation of logic for computing default values via `constexpr if`.

Containers that the user passes as arguments to KaMPIng are wrapped in a templated *DataBuffer* object, which handles ownership and modifiability transparently at compile time.[5] Subsequently, KaMPIng moves these data buffers to a templated result object, which is returned to the user and can be deconstructed using structured bindings (Section 6.4.2). As we do not copy but move the data, this imposes nearly zero overhead.

We extensively test KaMPIng for robustness. For this, we employ a test suite spanning over 4700 checks, distributed over 250 test case collections and 24 000 lines of test code – compared to 10 000 lines of code for the actual implementation. We compile and execute all unit and compilation failure tests four times: Using two C++ compilers (clang and GCC), as well as in debug and release mode. We use OpenMPI as our MPI implementation (Section 2.6). We ensure that KaMPIng only performs the expected MPI calls by instrumenting MPI's profiling interface, including cases in which KaMPIng invokes additional MPI calls in order

[5] Because this works with any STL-compliant container, KaMPIng also supports accelerator-aware MPI implementations directly. Pointers or containers (like `thrust::device_vector`) to device memory can be passed just like normal containers.

to automatically compute missing parameters. Further, we test our benchmark applications using OpenMPI and IntelMPI on the SuperMUC-NG and HoreKa HPC systems, as detailed in the following section.

6.5 Integrating KaMPIng into Real-World Applications

To highlight the usability of our library, we integrate KaMPIng into multiple (scientific) applications, ranging from sorting (sample sort and suffix sorting) over graph algorithms (Breadth First Search (BFS) and label propagation) to the phylogenetic interference tool RAxML-NG (Section 2.3). We run experiments to back our (near) zero overhead claim on up to 256 compute nodes of SuperMUC-NG[6], where each node is equipped with an Intel Skylake Xeon Platinum 8174 processor with 48 physical cores. The internal interconnect is an OmniPath network with 100 Gbit s^{-1} (Section 2.5). We compile with GCC 12.2.0 and Intel MPI 2021 using optimization level `-O3`.

Table 6.1 Lines of code for examples using KaMPIng vs. other MPI wrapper libraries.[7]

	MPI	Boost.MPI	RWTH-MPI	MPL	**KaMPIng**
vector allgather	14	5	5	12	1
sample sort	32	30	21	37	16
BFS	46	42	32	49	22

6.5.1 Distributed Sample Sort and Suffix Sort

We implement a textbook distributed sample sort [San+19] with KaMPIng (Figure 6.7) and all other previously discussed C++ MPI bindings (Section 6.3). We extract code that is common to all implementations into shared functions and format each source code with `clang-format` using the `Google` style template. We find, that the implementation using KaMPIng is 16 lines of code (LOC) long. In contrast, a plain-MPI implementation requires 32 LOC (Table 6.1). Further, we measure the runtimes of all implementations by sorting a distributed array with 10^6 64 bit integers per PE. We draw the values of the array uniformly at random and show that the KaMPIng implementation has no measurable runtime overhead over other bindings or plain-MPI (Figure 6.8). We experimentally verify that all variants arrive at the same (unique) suffix array and that the output of all variants is sorted.

Suffix Array Construction We also consider the more complex example of Suffix Array Construction from the field of text processing: Here, we sort all suffixes of a text lexicographically, that is, we compute the suffix array [MM93]. We implement two algorithms: DCX [KSB06] and Prefix Doubling [MM93]. Our KaMPIng-based DCX implementation requires 1264 LOC, whereas the plain-MPI implementation [Bin18a] requires 1396 LOC. The additional 9.5 % of code are accounted for by MPI boilerplate code, for instance, distributing send-counts for `MPI_Alltoallv` or the tedious construction of MPI types. Likewise, our

[6] We also conducted some experiments on the smaller HoreKa supercomputer. As the results obtained there are similar to our findings from SuperMUC-NG, we omit them here.

[6] The source codes for all three examples are available at `https://github.com/kamping-site/kamping-examples/`.

```
template<typename T>
void sort(std::vector<T>& data, MPI_Comm comm_) {
  using namespace std;
  Communicator comm(comm_);
  const size_t num_samples = 16 * log2(comm.size()) + 1;
  vector<T> lsamples(num_samples);
  sample(data.begin(), data.end(), lsamples.begin(),
        num_samples, mt19937{random_device{}()});
  auto gsamples = comm.allgather(send_buf(lsamples));
  sort(gsamples.begin(), gsamples.end());
  for (size_t i = 0; i < comm.size() - 1; i++) {
    gsamples[i] = gsamples[num_samples * (i + 1)];
  }
  gsamples.resize(comm.size() - 1);
  vector<vector<T>> buckets = build_buckets(data, gsamples);
  data.clear();
  vector<int> scounts;
  for (auto &bucket : buckets) {
    data.insert(data.end(), bucket.begin(), bucket.end());
    scounts.push_back(bucket.size());
  }
  data = comm.alltoallv(
    send_buf(data), send_counts(scounts)
  );
  sort(data.begin(), data.end());
}
```

Figure 6.7 Distributed sample sort using KaMPIng.

KaMPIng-based Prefix Doubling implementation requires 163 LOC. An existing plain-MPI implementation of the same algorithm [FK19] requires 426 LOC (not counting the 1442 LOC for wrapped MPI functionality used by the plain-MPI implementation). Even when using the high-level distributed programming framework Thrill, implementing the Prefix Doubling algorithm still requires 266 LOC [BGK18].

6.5.2 Graph Algorithms

Currently, most state-of-the-art HPC platforms are primarily designed to perform well on numerical applications with regular data access and communication patterns. However, data-intensive irregular workloads occur, for example, in data mining [KF13], or brain analysis [Rin+16]. In these applications, operations on graphs are common and necessitate efficient distributed graph algorithms – either for network analysis or as building blocks for more complex applications [Dav+85; Höt+14; SRM16].

Breadth First Search To demonstrate the applicability of KaMPIng in this setting, we provide a simple distributed implementation of a Breadth First Search (BFS). We assume the input graph to be distributed among the PEs with each PE holding a subset of the vertices including their incident edges. This subgraph is represented as an adjacency array. For each vertex v, the BFS returns the distance (number of hops) between the source vertex `s` and v. Here, we use KaMPIng's utility function `with_flattened(...)`, which flattens a container of nested destination-message pairs by transforming it into a contiguous range, while also providing send-counts.

We implement the distributed BFS algorithm using all C++ MPI bindings discussed in Section 6.3. These implementations only differ with respect to the logic used for message

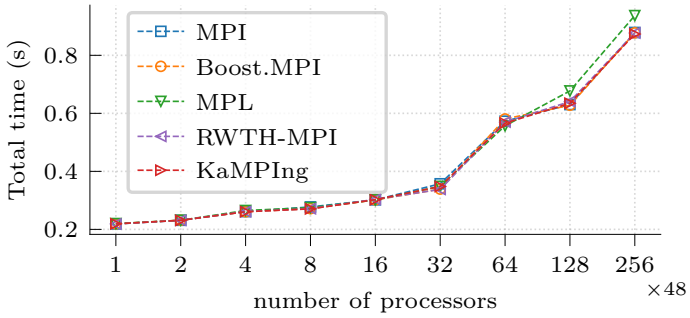


Figure 6.8 Running time of sample sort using different MPI bindings.

exchanges and completion checking. We implement this logic using KaMPIng in only 22 lines of code, whereas plain-MPI requires 46 LOC, and the second most concise binding, RWTH-MPI, requires 32 LOC (Table 6.1).[8]

We evaluate the implementations' running times for different simulated graph families (Figure 6.9). We find, that KaMPIng introduces no additional runtime overhead compared to plain-MPI. Further, KaMPIng provides optimized collective operations, which scale better than `MPI_Alltoallv` on some graph families and are easy to use. We discuss the obtained speedups in Section 6.6.1. When not using these optimized collectives for comparability, the runtime of our KaMPIng implementation is on par with those of the plain-MPI implementation. This also holds for the RWTH-MPI and Boost.MPI (omitted in the plot) implementations. MPL (also omitted in the plot) internally uses `MPI_Alltoallw` for all-to-all exchanges [Gho+21] and is thus (considerably) slower than MPI on all inputs. We experimentally verify that all variants arrive at the same BFS level for all graph nodes.

Graph Partitioning As a more complex showcase, we integrated KaMPIng into the state-of-the-art distributed multilevel graph partitioner dKaMinPar [SS23], comprising approximately 30 000 LOC and including its own MPI abstraction layer, which features specialized graph-specific communication primitives. The partitioner iteratively clusters and contracts the input graph via size-constrained label propagation, shrinking the graph until a certain threshold. Next, dKaMinPar approximates the clustering on this contracted graph and subsequently re-expands it to refine the solution via further heuristics. Due to the size of the dKaMinPar project, we only consider the graph-shrinking step here. Again, we compare an implementation based on dKaMinPar's application-specific MPI abstraction layer, a plain-MPI based implementation, and a KaMPIng-based implementation. We extract the shared code of all implementations (202 LOC) to shared functions and only consider the MPI-heavy part of the algorithm's implementation. We find, that the plain-MPI based implementation (154 LOC) is roughly 17.5 % larger than the KaMPIng-based implementation (127 LOC), which in turn is 16.5 % larger than the implementation based on dKaMinPar's specialized abstraction layer (106 LOC). We observe the same running times for all these variants. We experimentally verify that all variants arrive at the same cut and balance – and thus most likely the same partitioning.

[8] `https://github.com/kamping-site/kamping-examples/tree/main/include/bfs`

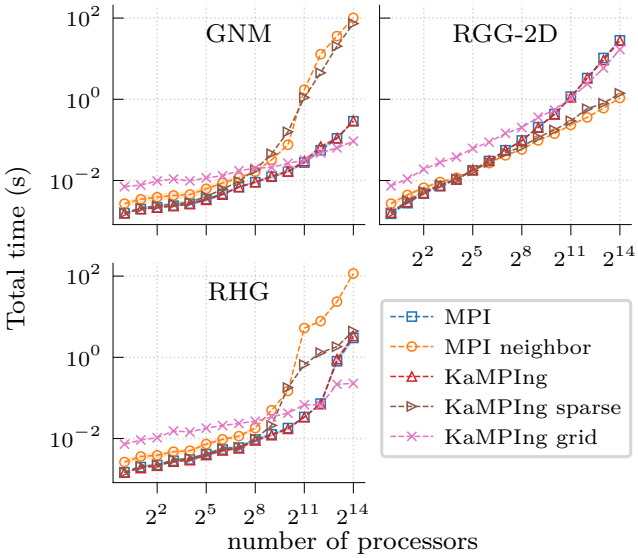


Figure 6.9 Running time of Breadth First Search (BFS) implementations using plain-MPI or KaMPIng on graphs from the Erdős-Rényi (GNM), Random Geometric Graph (RGG), and Random Hyperbolic Graph (RHG) families. The *sparse* and *grid* all-to-all implementations are optimized for irregular sparse all-to-all message exchanges and described in Section 6.6.1. The *MPI neighbor* variant employs MPI's virtual topology based neighbor communication (Section 6.2).

6.5.3 Integrating KaMPIng into RAxML-NG

We consider the phylogenetic inference tool RAxML-NG [Koz+19] (Section 2.3) as our largest real-world application benchmark. RAxML-NG is written in C++ and uses a custom non-trivial abstraction layer over `pthreads` + MPI parallelism with over 700 LOC. We use KaMPIng to substantially simplify the MPI part of this abstraction layer; demonstrating that KaMPIng can easily be integrated even in large and well-established scientific codes that use hybrid parallelization (for an example, see Figure 6.10). If KaMPIng had been available at the time, the RAxML-NG developers would have never needed to write, unit-test, maintain, and document over a hundred lines of complex code.[9]

We empirically verified that replacing the abstraction layer with KaMPIng does not incur a measurable performance overhead even though RAxML-NG issues 700 MPI calls per second.[10] The binary's size does not increase substantially (only by 2.5 %), while compilation time increases from 1:15 min to 1:30 min. We experimentally verify that all variants arrive at the same tree topology an yield identical Log-Likelihood scores (Section 2.3).

6.6 Towards A Basic Toolbox for Distributed Computing

A standard library of efficient distributed (communication) algorithms and data structures would substantially ease the development of distributed-memory codes. However, incorporating this functionality in KaMPIng's core would make it overly complex. Thus, we decided to

[9] In RAxML-NG, MPI and application code are heavily intertwined, yielding a fair and exact LOC comparison infeasible.

[10] The mean runtimes times are less than one standard deviations apart.

```
// Before. The self-written mpi_broadcast(...) wrapper and
// serialization/deserialization of data are not shown.
template<typename T>
static void mpi_broadcast(T& obj) {
  if (_num_ranks > 1) {
    size_t size = master() ?
      BinaryStream::serialize(
        _parallel_buf.data(),
        _parallel_buf.capacity(),
        obj)
      : 0;
    mpi_broadcast((void *) &size, sizeof(size_t));
    mpi_broadcast((void *) _parallel_buf.data(), size);
    if (!master()) {
      BinaryStream bs(_parallel_buf.data(), size);
      bs >> obj;
    }
  }
}

// After. KaMPIng provides all required functionality.
template <typename T>
static void mpi_broadcast(T &obj) {
  if (_num_ranks > 1) {
    _comm->bcast(send_recv_buf(as_serialized(obj)));
  }
}
```

Figure 6.10 Example of a function in the RAxML-NG parallelization abstraction layer simplified substantially using KaMPIng. We are able to replace custom serialization logic entirely.

develop and make available multiple library extensions as KaMPIng plugins (Section 6.4.6). This includes an STL-like distributed sorter (Section 6.5.1), all-to-all operations that are optimized for sparse and irregular communication patterns, and an abstraction over MPI's failure-tolerance mechanism, as well as reproducible reduction operations.

6.6.1 Sparse and Low-Latency All-To-All communication

All-to-all exchanges are a common communication pattern in distributed computing. However, there is a large algorithmic design space for all-to-all communication, ranging from algorithms with near-optimal communication volume but latency at least linear in the number of PEs to algorithms following a hypercube communication scheme with logarithmic latency but a communication volume that increases by a logarithmic factor [San+19].

In KaMPIng's `GridCommunicator` plugin, we implement communication in a two-dimensional grid [KKV03], thereby reducing latency at the cost of communication volume. To this end, we organize the PEs in a virtual two-dimensional grid and route messages along the rows and columns of this grid in two hops to their destination. This yields a message start-up latency (Section 2.5) in $\mathcal{O}(\sqrt{p})$, where p denotes the number of PEs in the communicator. This enables hardware-agnostic latency reduction with asymptotic guarantees, in contrast to the variants provided by most MPI implementations.

MPI's standard collectives have not been designed for *sparse* communication patterns. That is, an all-to-all communication patterns where only $\mathcal{O}(p)$ out of the p^2 messages are non-empty. `MPI_Alltoallv`, for instance, requires a send-counts parameter consisting of an array with one entry for each PE of the communicator, yielding a time complexity that is linear in the communicator size. To mitigate this problem for static communication patterns,

MPI-3.0 adds *neighborhood* collectives, which allow the user to perform `MPI_Alltoall(v)` and `MPI_Allgather(v)` on a previously defined (sparse) graph topology. However, for applications and algorithms with rapidly changing communication partners, for instance, (dynamic) graph algorithms, the overhead of defining a new communication graph topology every few all-to-all exchanges may impose a substantial overhead. KaMPIng's `SparseAlltoall` plugin offers a lightweight alternative for these scenarios and accepts a set of destination-message pairs as argument. For the actual data exchange, the plugin uses the *NBX* algorithm for sparse all-to-all communication by Hoefler *et al.* [HSL10].

Both techniques for improving irregular sparse exchanges specifically improve the scalability of distributed graph algorithms [SS23; SU23]. Figure 6.9 shows an evaluation of our different all-to-all strategies using the weak-scaling BFS benchmark introduced earlier on three different graph families [Fun+19], where each PE holds 2^{12} vertices and 2^{15} edges. Erdős-Rényi graphs exhibit almost no locality (most edges cross PE-boundaries) but a small diameter, whereas Random Geometric Graphs (RGGs) are highly local but exhibit a high diameter. Regarding locality, Random Hyperbolic Graphs (RHGs) range somewhere in between and also exhibit a small diameter. In contrast to Erdős-Rényi graphs and Random Geometric Graphs (RGGs), they contain high-degree vertices.

In Figure 6.9, we compare different algorithms for the actual frontier exchange in each BFS step: `MPI_Alltoallv` (*MPI*, *KaMPIng*), KaMPIng's sparse all-to-all plugin, KaMPIng's grid all-to-all plugin, and `MPI_Neighbor_alltoallv` (*MPI neighbor*). For Random Hyperbolic Graphs (RHGs), and – less pronounced – for Erdős-Rényi graphs, our grid all-to-all algorithm scales best. Grid all-to-all also outperforms the built-in `MPI_Alltoallv` on RGGs. Due to their high diameter and locality, a competitive performance on RGGs can only be achieved via sparse communication. KaMPIng's sparse all-to-all approach is only slightly slower than `MPI_Neighbor_alltoallv`. Note that when rebuilding MPI's communication graph before each frontier exchange, which simulates dynamic communication patterns to some extent, `Neighbor_alltoallv` does not scale.

6.6.2 User-Level Failure Mitigation

With the increasing number of processors in HPC systems, hardware failures represent one of the major challenges for upcoming exascale systems [SDM10] (Section 2.7). The upcoming MPI 5.0 standard enables development of software that can recover from such failures [Bla+13] (Section 2.9). We implement an abstraction layer over the failure-mitigating mechanisms discussed in Section 2.9. First, we add a custom error handling hook, which notifies the user of a PE-failure via an idiomatic C++ exception instead of using return codes (Figure 6.11). Next, the user can notify all PEs of a failure – which might be known only to a subset of PEs – by *revoking* the communicator using `comm.revoke()`. The application can subsequently obtain a list of failed PEs via `comm.get_failed()` or create a new communicator excluding the failed PEs using `comm.shrink()`. Note that we also implement the other functions of the User Level Failure Mitigation (ULFM) standard proposal [ULFM] that we omitted here.

6.6.3 Bit-Reproducible Distributed Reduction Under Varying Core-Counts

Computational reproducibility is considered a cornerstone for validating scientific claims [IT18] (Section 2.10). Thus, one major challenge in distributed computing is to ensure that the results of a computation are bit-reproducible across different runs using varying PE-counts. However, IEEE 754 [IEEE754] floating-point math is not associative because of rounding errors. Thus, the order of operations, which often depends on the PE-count, can influence

```
try {
  comm.allreduce(/* ... */);
} catch ([[maybe_unused]] MPIFailureDetected const& _) {
  // Notify all other PEs in the communicator of the failure.
  if (!comm.is_revoked()) {
      comm.revoke();
  }
  // Create a new communicator containing only the surviving PEs.
  comm = comm.shrink();
}
```

Figure 6.11 Handling a PE failure using KaMPIng's failure-mitigation plugin.

the result (Section 2.10). To this end, we provide a KaMPIng plugin that implements a bit-reproducible distributed reduction algorithm (Chapter 5). Analogous to the "normal" KaMPIng `reduce`, this plugin supports plain-MPI constants, function pointers, and lambda functions as operations, assuming only associativity, a necessary requirement for parallelism. We believe that the availability of this convenient, off-the-shelf library method will encourage more authors to ensure that their applications yield bit-reproducible results.

6.7 Conclusion and Future Work

We introduce a novel set of C++ MPI bindings and our open-source[11], (near) zero-overhead implementation, KaMPIng. Through configurable inference of parameter defaults, fine-grained memory allocation control, enhanced safety guarantees, and a flexible plugin system, KaMPIng enables both, rapid prototyping, and careful fine-tuning of distributed algorithms. We demonstrate this claim via a variety of benchmarks from different application domains, including, for example, distributed sorting and graph algorithms, as well as the phylogenetic inference tool RAxML-NG (Section 6.5.3).

KaMPIng is extensively tested and currently used in multiple research projects, for example, suffix array construction [Haa+24], distributed string sorting [Kur+24], maximum weighted independent set kernelization [Bor25], as well as distributed list-ranking and tree-routing [Wei24]. Further, KaMPIng covers MPI's collectives, blocking, and non-blocking point-to-point operations (Section 2.6) – the most commonly used MPI features [Lag+19]. In the future, we plan to extend KaMPIng to cover most of the MPI standard, while also building a basic algorithmic toolbox on top of KaMPIng. We hope that this will facilitate rapid prototyping with short feedback loops and iterative improvements of distributed algorithms with a strong focus on runtime performance. To this end, we will also further elicit the needs of the MPI community and are currently involved in the MPI language binding working group in order to develop novel, standardized C++ bindings for MPI and establish KaMPIng as a prototypical reference implementation. Further, we are currently generalizing the indirection patterns employed in all-to-all primitives to higher dimensions, while also incorporating message aggregation. These improvements are applicable in request-replay patterns when reading from globally distributed data, and algorithms with highly-irregular communication without hard synchronization. We are implementing these building blocks on top of KaMPIng and will integrate them into future library releases.

We are currently developing distributed containers, with the aim of enabling lightweight

[11] `https://github.com/kamping-site/kamping`; LGPL

bulk parallel computation as in MapReduce [DG04] or Thrill [Bin+16], while not locking the programmer into the walled garden of a particular framework.

In summary, we strive to establish KaMPIng as a stable core for an entire ecosystem of future general-purpose distributed algorithms and applications.

7 Conclusion and Future Work

Attribution: I am the sole author of the *text* in this chapter. Attributions of the *work* referenced herein can be found in the respective chapters.

In this thesis, we address four major challenges posed by the exponential growth of available genomic data to the scalability of evolutionary bioinformatics applications (Section 1.1). In order to handle this exponential growth, we require highly parallel and scalable codes as well as work-efficient algorithms. As increased parallelism causes programs to more frequently experience hardware faults, we require our algorithms to support rapid recovery after such a fault. Further, to strengthen scientific claims, we require algorithms that yield bit-identical results across different hardware environments, for instance, using a distinct number of CPU cores or architectures. Finally, software libraries that encapsulate the abstractions we utilize to manage the growing complexity of scientific codes facilitate the rapid development of these applications.

7.1 The Algorithmic Challenge

In Chapter 3, we introduce a novel data structure for storing sets of genealogical trees and their associated genomic sequences that enables the memoization of intermediate results. The current state-of-the-art data structure compresses these trees via edit operations when moving from one tree to the next along the genome. In contrast, our data structure, `gfkit`, compresses these trees into a Directed Acyclic Graph, where each node represents a unique subtree of the input. This allows us to memoize intermediate results and yields speedups over the state-of-the-art tool ranging between 2.1 and 11.2 (median 4.0) when computing important population genetics statistics such as the Allele Frequency Spectrum, Patterson's f, the Fixation Index, Tajima's D, the pairwise Lowest Common Ancestor (LCA), and others on empirical as well as simulated datasets. On LCA queries with more than two samples as input, `gfkit` scales asymptotically better than the state-of-the-art implementation, and is thus up to two orders of magnitude faster in our experiments. Further, as more genomes are being sequenced, future datasets will contain a multiple of today's samples. On a simulated human dataset with 640 000 samples, we observe speedups (and running times) that are comparable to the largest existing empirical datasets.

Our advancements will boost the development of novel analyses and models in the field of population genetics, and improve scalability to accommodate the exponentially-growing genomic datasets. For instance, `gfkit` might be used for developing appropriate optimization criteria and techniques for automatically determining subpopulations.

7.2 The Hardware-Failure Mitigation Challenge

In Chapter 4, we improve the mitigation of hardware failures in High Performance Computing (HPC). Since it is impractical to either, request replacement for failed CPU cores or to maintain spares, applications must continue execution with the remaining cores (*shrinking recovery*). Thus, during failure-recovery, applications must re-distribute their workload and

reload the respective input data to the remaining cores. We propose a novel algorithmic framework and C++ implementation, ReStore, which is, to the best of our knowledge, the first framework that supports shrinking recovery from memory to mitigate hardware failures in Message Passing Interface (MPI) programs. This allows the applications to utilize all CPU cores for computation rather than reserving some as spares to replace failed cores.

By storing data in memory with an appropriate data distribution and replication scheme, recovery is substantially faster compared to checkpointing approaches that rely on the parallel file system. We evaluate ReStore both, in an isolated environment, and with real applications. Our experiments show loading times for lost data in the range of milliseconds on up to 24 576 cores. Our data distribution ensures a low data loss probability and allows for rapid data recovery by balancing the bottleneck communication volume and the bottleneck number of messages. Additionally, we analyze the probability of irrecoverable data loss and propose a data distribution strategy to easily restore lost replicas after a failure.

ReStore is compatible with any HPC application using MPI 5.0 – which introduces fault-tolerance mechanisms. For example, by employing ReStore, we improve the recovery performance of FT-RAxML-NG [Hüb+21b; Koz+19] by up to two orders of magnitude. In summary, ReStore enables scientific applications to recover from hardware failures in milliseconds, necessitating neither replacement nor spare cores.

7.3 Reproducibility Challenge

Maximum-likelihood phylogenetic inference tools such as RAxML-NG use heuristics to search for the phylogenetic tree that best explains the shared evolutionary history of the input genomic sequences (Sections 2.1 and 2.3). However, due to differing rounding errors induced by the re-association of the IEEE 754 floating-point computations performed during a distributed all-reduce operation, these algorithms yield different results under varying CPU core counts. Concretely, and supporting the findings by Shen *et al.* [She+20a], we observe that the level of parallelism contributes to diverging phylogenetic tree searches in 31 % of 10 130 empirical datasets (Chapter 5). Moreover, 8 % of these datasets yield trees that are significantly worse than the best found tree (AU-test, $p < 0.05$) depending on the parallelization level. To alleviate this issue, we implement a variant of the widely-used phylogenetic inference tool RAxML-NG that yields bit-reproducible results under varying core counts. This variant is merely 0 to 6.7 % (median 0.93 %) slower than non-reproducible RAxML-NG on up to 768 cores.

We also introduce ReproRed, a bit-reproducible parallel reduction algorithm, which specifies the order of reduction operations independent of the communication patterns. Thus, ReproRed is applicable to arbitrary associative reduction operators – in contrast to its competitors, which are confined to summation. This allows us to integrate ReproRed into a open-source MPI wrapper library that enables users to pass arbitrary reduction operators, such as C++ lambdas or function pointers. Furthermore, ReproRed only exchanges the theoretical minimum number of messages, overlaps communication with computation, and utilizes a fast base-case for conducting local reductions.

We find that ReproRed is able to all-reduce (via a subsequent broadcast) 4.1×10^6 operands on 48 to 768 cores in 19.7 to 48.61 µs. Thus, ReproRed induces a slowdown of only 13 to 93 % over an non-reproducible all-reduce algorithm and of only 8 to 25 % over an non-reproducible Reduce-Bcast algorithm. ReproRed outperforms the state-of-the-art reproducible all-reduce algorithm ReproBLAS when reducing over more than 10 000 elements per core.

In conclusion, we re-asses the issue of non-reproducibility in phylogenetic inference and present the, to the best of our knowledge, first bit-wise reproducible parallel phylogenetic inference tool. Additionally, we introduce and evaluate a bit-reproducible distributed-memory reduction algorithm, ReproRed. In contrast to competitors, ReproRed supports arbitrary associative reduction operations. This allows us to integrate ReproRed into an open-source MPI-wrapper library, which facilitates the transition to bit-reproducible code.

7.4 Software Complexity Challenge

In Chapter 6, we propose novel C++ MPI language bindings and the accompanying implementation, KaMPIng, that is designed to cover all abstraction levels from low-level MPI calls to convenient Standard Template Library style bindings with automatic inference of missing parameters, where feasible. Through this automatic computation of missing parameters, fine-grained memory allocation control, enhanced safety guarantees, and a flexible plugin system, KaMPIng enables both, rapid prototyping and careful fine-tuning of distributed algorithms. We utilize template-metaprogramming to ensure that the compiler only generates code paths that are necessary to compute the missing parameters for a specific function invocation, resulting in (near) zero-overhead bindings. Moreover, KaMPIng supports MPI's collectives, and both, blocking and non-blocking point-to-point operations (Section 2.6) – which are among the most commonly used MPI features [Lag+19].

KaMPIng is extensively tested and currently used in multiple research projects, for example, suffix array construction [Haa+24], distributed string sorting [Kur+24], maximum weighted independent set kernelization [Bor25], as well as distributed list-ranking and tree-routing [Wei24]. Using multiple application benchmarks, from sorting algorithms to phylogenetic interference with RAxML-NG, we demonstrate that KaMPIng provides a foundation for a future distributed standard library. Additionally, we implement a bit-reproducible distributed-memory reduction algorithm and integrate MPI's failure-mitigation bindings into KaMPIng.

7.5 Future Work

We now outline several high-level directions of future work. Note that each chapter includes more specific directions of future work.

As outlined in Chapter 6, we are currently extending KaMPIng with distributed containers to facilitate lightweight bulk parallel computation as in MapReduce [DG04] or Thrill [Bin+16]. Many of these functions, for instance, the `map` and `reduce` of MapReduce are pure [Hua+12; SR05], that is, they are side effect free and consistently return identical result when invoked with identical arguments. Pure functions allow for automatic parallelization [SS16; Süß+20], load-balancing [Bin+16; Bin18b], fault-tolerance [Hes23; Mem+16], and possibly avoiding some floating-point reproducibility issues. Consequently, future work could involve designing KaMPIng's bulk parallel operations to incorporate automatic fault-tolerance using ReStore and enhance reproducibility using ReproRed.

To obtain bit-reproducible results across distinct implementations, scientific communities could establish a standardized sequence for common calculations, for instance, the statistics in population genetics discussed in Chapter 3. For example, bit-reproducibility in computing sequence *diversity*,[1] could be achieved by employing sufficiently large integer variables to

[1] The probability that two random sequences differ at a random site (both chosen uniformly at random).

accommodate an intermediate result and by specifying the used floating point data-types, for instance, as 64 bit IEEE 754. Additionally, the average value across the genome should be calculated utilizing a pair-wise summation over the per-locus values. This fixes the operation order while enabling parallel computation (Chapter 5).

A further substantial challenge to the scalability of evolutionary bioinformatics algorithms not discussed in this thesis is the need to incrementally update existing data structures with new data. For instance, when new genomes are sequenced and shall be integrated into a genealogical forest or tree sequence, one must avoid re-building the entire data structure in order to ensure scalability. Therefore, it is essential to expand tree sequence inference algorithms as well as other bioinformatics data structures to support incremental updates.

7.6 Conclusion

In conclusion, in this thesis, we design, implement, and evaluate a work-efficient data structure for statistical queries on population genetics datasets and a novel scalable in-memory replicated storage for failure-tolerant applications. Further, we present a distributed-memory reduction algorithm that guarantees reproducible results, irrespective of the core-count, and that is applicable to arbitrary associative reduction operators. Moreover, we implement the first bit-reproducible parallel phylogenetic tree inference tool. We also develop high-level C++ bindings for MPI, thereby enabling a faster, less error-prone development of distributed applications, including access to MPI's failure-mitigation mechanisms and our reproducible parallel reduce implementation. While motivated by challenges in evolutionary bioinformatics, most of our results are also applicable to other HPC domains. Our work allows researchers to analyze the exponentially growing phylogenetic and population genetic datasets, to perform novel, work-intensive analyses, and to publish reproducible results. It also facilitates the development of complex distributed-memory algorithms through abstractions.

Appendix

Acronyms

ABFT Algorithm-Based Failure-Tolerance. 17
AFS Allele Frequency Spectrum. 4, 14, 109
AU Approximately Unbiased. v, x, 5, 63, 65, 68, 74, 110

BFS Breadth First Search. 101
BLAS Basic Linear Algebra Subprograms. 68
BLCR Berkeley Lab Checkpoint/Restart. 18

DAG Directed Acyclic Graph. v, vi, ix, 4, 23, 109
DNA deoxyribonucleic acid. 10, 12

FLOPS floating-point operations per second. 71
FMA Fused Multiply-and-Add. 20

GRG Gene Recombination Graph. 27

HPC High Performance Computing. vii, 3, 9, 41, 42, 66, 84, 109

IDL Irrecoverable Data Loss. 49

LCA Lowest Common Ancestor. vi, x, 4, 10, 23–26, 71, 109
LCG Linear Congruential Generator. 51
LLH Log-Likelihood. 13, 67, 104
LOC lines of code. 101

ML Maximum-Likelihood. 14
MPI Message Passing Interface. v, vi, ix, xi, 3, 9, 41, 43, 63, 83, 84, 110
MPL Message Passing Library. 90
MRCA Most Recent Common Ancestor. 15
MSA Multiple Sequence Alignment. 13, 74, 80

PE Processing Element. 15, 42, 64, 87
PFS parallel file system. vi, 3, 19, 41, 42, 110
PGAS Partitioned Global Address Space. 45

RAII Resource Acquisition Is Initialization. 84, 91, 97
RGG Random Geometric Graph. 104, 106
RHG Random Hyperbolic Graph. 104, 106
RMA Remote Memory Access. 45
RNA ribonucleic acid. 10, 14

SFINAE Substitution Failure is Not an Error. 100
SGDP Simons Genome Diversity Project. 32
SIMD Single Instruction Multiple Data Stream. 20, 65

SRA Sequence Read Archive. 2
STL Standard Template Library. vi, xi, 6, 84, 85, 111

TGP Thousand Genomes Project. 6, 25

ULFM User Level Failure Mitigation. 19, 43, 106

List of Figures

List of Tables

Publications and Supervised Theses

In Conference Proceedings

- T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, D. Seemaier, and P. Sanders. "KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI". in: *High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pages 689–709. DOI: 10.1109/sc41406.2024.00050
- D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, and T. N. Uhl. "Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI". in: *36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. volume 31. SPAA '24. ACM Press, 2024, pages 289–291. DOI: 10.1145/3626183.3660260
- L. Hübner and A. Stamatakis. "Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests". In: *24th Workshop on Algorithms in Bioinformatics (WABI)*. edited by S. P. Pissis and W. Sung. Volume 312. Leibniz International Proceedings in Informatics (LIPcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pages 1–22. ISBN: 978-3-95977-340-9. DOI: 10.4230/lipics.wabi.2024.5
- J. Haag, L. Hübner, A. M. Kozlov, and A. Stamatakis. "The Free Lunch is not over yet – Systematic Exploration of Numerical Thresholds in Maximum Likelihood Phylogenetic Inference". In: *Bioinformatics Advances* 3.1 (2023). Edited by A. Ouangraoua. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbad124
- D. Hespe, L. Hübner, L. Hübschle-Schneider, P. Sanders, and D. Schreiber. "Scalable Discrete Algorithms for Big Data Applications". In: *High Performance Computing in Science and Engineering '21*. Springer, 2023, pages 439–449. ISBN: 9783031179372. DOI: 10.1007/978-3-031-17937-2_27
- L. Hübner, D. Hespe, P. Sanders, and A. Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *12th IEEE/ACM Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE Press, 2022, pages 24–35. DOI: 10.1109/ftxs56515.2022.00008
- I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. "Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning". In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '25)*. IEEE Press, 2019, pages 175–184. DOI: 10.1109/ipdpsw.2019.00038

Journal Articles

- L. Hübner, A. M. Kozlov, D. Hespe, P. Sanders, and A. Stamatakis. "Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". in: *Bioinformatics* 37.22 (2021). Edited by R. Schwartz, pages 4056–4063. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab399
- B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mamais, M. A. Kozlov, P. Pavlidis, D. Paraskevis, and A. Stamatakis. "Phylogenetic Analysis of SARS-CoV-2 Data Is Difficult". In: *Molecular Biology and*

Evolution 38.5 (2020). Edited by H. Malik, pages 1777–1791. ISSN: 1537-1719. DOI: 10.1093/molbev/msaa314

Technical Reports

- D. Hespe, L. Hübner, C. Mercatoris, and P. Sanders. *Scalable Fault-Tolerant MapReduce*. Technical report. Karlsruhe Institute of Technology, 2024. DOI: 10.48550/arxiv.2411.16255
- L. Hübner and A. Stamatakis. *Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests*. Technical report. Karlsruhe Institute of Technology, 2024. DOI: 10.1101/2024.05.23.595533
- T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, C. Stelz, and P. Sanders. *KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI*. technical report. Karlsruhe Institute of Technology, 2024, pages 1–21. DOI: 10.1109/sc41406.2024.00050
- L. Hübner, D. Hespe, P. Sanders, and A. Stamatakis. *ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms*. Technical report. Karlsruhe Institute of Technology, 2022. DOI: 10.48550/arxiv.2203.01107
- J. Haag, L. Hübner, A. M. Kozlov, and A. Stamatakis. *The Free Lunch is not over yet – Systematic Exploration of Numerical Thresholds in Phylogenetic Inference*. Technical report. Heidelberg Institute for Theoretical Studies, 2022. DOI: 10.1101/2022.07.13.499893
- L. Hübner, A. M. Kozlov, D. Hespe, P. Sanders, and A. Stamatakis. *Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG*. technical report. Karlsruhe Institute of Technology, 2021. DOI: 10.1101/2021.01.15.426773
- B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mmai, M. A. Kozlov, P. Pavlidis, D. Paraskevis, and A. Stamatakis. *Phylogenetic Analysis of SARS-CoV-2 Data is Difficult*. Technical report. Heidelberg Institute for Theoretical Studies, 2020. DOI: 10.1101/2020.08.05.239046
- I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. *Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning*. Technical report. Karlsruhe Institute of Technology, 2019. DOI: 10.1101/579318

Theses

- L. Hübner. "Load-Balance and Fault-Tolerance for Massively Parallel Phylogenetic Inference". Master's thesis. Karlsruhe Institute of Technology, 2020
- L. Hübner. "Analysis of Epigenetic Modulators Across Multiple Cancer Types". Bachelor's Thesis. Heidelberg University, 2017
- L. Hübner. "Coloring Complex Networks". Bachelor's Thesis. Karlsruhe Institute of Technology, 2014

Supervised Theses

- J. Hengstler. “Re-Engineering Genomic Tree Sequence Inference Algorithms”. Master’s Thesis. Karlsruhe Institute of Technology, 2024
- D. Siebelt. “Faster Sorting of Aligned DNA-Read Files”. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2024
- G. Zeitzmann. “Lightweight Checkpointing and Recovery for Iterative Converging Algorithms”. Master’s Thesis. Karlsruhe Institute of Technology, 2023
- C. Stelz. “Core-Count Independent Reproducible Reduce”. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2022
- C. Mercatoris. “Scalable Decentralized Fault-Toleranc MapReduce for Iterative Algorithms”. Master’s Thesis. Karlsruhe Institute of Technology, 2021
- J. Haag. “Empirical Numerical Properties of Maximum Likelihood Phylogenetic Inference”. Master’s Thesis. Karlsruhe Institute of Technology, 2021

Usage of Generative Models

In accordance with the statement of the Deutsche Forschungsgemeinschaft[1] on the 21st of September 2023 regarding the usage of generative models for text and image generation, we want to indicate that generative models were used for the following purposes: Part of the code accompanying this thesis was developed using GitHub Copilot (based on GPT-3.5, that is, pre-ChatGPT) for (a) assisting writing a small subset of unit tests for software artifacts, (b) assisting writing a small subset of inline code documentation (Doxygen), and (c) auto-completion at sub-line and line level. All generated unit tests, documentation, and auto-completion have been verified manually by the authors. Further, no code generation or other generative model has been used for interface, data structure, or algorithm design.

In the publication on which Chapter 6 is based, the grammatical correctness of a small number of sentences were verified via ChatGPT-3. Moreover, we utilized GPT-4o when writing Chapters 3, 5, and 7 to propose linkage words, stronger verbs, and shorter formulations. All proposed edits have been manually evaluated and accepted by the authors. No image in this thesis was generated using generative models.

[1] https://www.dfg.de/de/aktuelles/neuigkeiten-themen/info-wissenschaft/2023/info-wissenschaft-23-72

Bibliography

[Aar+15] A. A. Aarts, J. E. Anderson, C. J. Anderson, P. R. Attridge, et al. "Estimating the Reproducibility of Psychological Science". In: *Science* 349.6251 (2015). ISSN: 1095-9203. DOI: 10.1126/science.aac4716 (cited on page 19).

[AAZ19] M. Adamek, M. Alanjary, and N. Ziemert. "Applied Evolution: Phylogeny-Based Approaches in Natural Products Research". In: *Natural Product Reports* 36.9 (2019), pages 1295–1312. ISSN: 1460-4752. DOI: 10.1039/c9np00027e (cited on page 11).

[ADN20] P. Ahrens, J. Demmel, and H. D. Nguyen. "Algorithms for Efficient Reproducible Floating Point Summation". In: *ACM Transactions on Mathematical Software* 46.3 (2020), pages 1–49. ISSN: 0098-3500. DOI: 10.1145/3389360 (cited on pages 68, 76–78).

[AFH14] A. Arteaga, O. Fuhrer, and T. Hoefler. "Designing Bit-Reproducible Portable High-Performance Applications". In: *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2014, pages 1235–1244. DOI: 10.1109/ipdps.2014.127 (cited on pages 64, 68, 69).

[Aga+04] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. "Adaptive Incremental Checkpointing for Massively Parallel Systems". In: *18th ACM International Conference on Supercomputing (ICS)*. Edited by P. Feautrier, J. R. Goodman, and A. Seznec. ACM Press, 2004, pages 277–286. DOI: 10.1145/1006209.1006248 (cited on pages 18, 19, 42, 44).

[AHE18] R. A. Ashraf, S. Hukerikar, and C. Engelmann. "Shrink or Substitute: Handling Process Failures in HPC Systems using In-Situ Recovery". In: *26th Euromicro Parallel, Distributed and Network-Based Processing (PDP)*. 2018 (cited on pages 18, 44).

[AHU73] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. "On Finding Lowest Common Ancestors in Trees". In: *5th ACM Symposium on Theory of Computing (STOC)*. Edited by A. V. Aho, A. Borodin, R. L. Constable, R. W. Floyd, M. A. Harrison, R. M. Karp, and H. R. Strong. STOC '73. ACM Press, 1973, pages 253–265. DOI: 10.1145/800125.804056 (cited on page 15).

[AKS14] A. J. Aberer, K. Kobert, and A. Stamatakis. "ExaBayes: Massively Parallel Bayesian Tree Inference for the Whole-Genome Era". In: *Molecular Biology and Evolution* 31.10 (2014), pages 2553–2556. DOI: 10.1093/molbev/msu236 (cited on page 57).

[Alb+22] B. Alberts, R. Heald, A. D. Johnson, D. Morgan, M. C. Raff, K. Roberts, et al. *Molecular Biology of the Cell*. eng. Seventh international edition. New York, NY: W. W. Norton & Company, 2022. ISBN: 9780393884852 (cited on pages 10, 12).

[Ali+16] M. M. Ali, P. E. Strazdins, B. Harding, and M. Hegland. "Complex Scientific Applications Made Fault-Tolerant with the Sparse Grid Combination Technique". In: *Journal of High Performance Computing Applications* 30.3 (2016), pages 335–359. DOI: 10.1177/1094342015628056 (cited on pages 17, 19, 43).

[All+24] M. E. Allentoft, M. Sikora, A. Fischer, K.-G. Sjögren, A. Ingason, et al. "100 Ancient Genomes Show Repeated Population Turnovers in Neolithic Denmark". In: *Nature* 625.7994 (2024), pages 329–337. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06862-3 (cited on pages 12, 24).

[All64] A. C. Allison. "Polymorphism and Natural Selection in Human Populations". In: *Cold Spring Harbor Symposia on Quantitative Biology* 29.0 (1964), pages 137–149. ISSN: 1943-4456. DOI: 10.1101/sqb.1964.029.01.018 (cited on pages 1, 12, 24).

[Als+04] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. "Nearest Common Ancestors: A Survey and a New Algorithm for a Distributed Environment". In: *Theory of Computing Systems* 37.3 (2004), pages 441–456. DOI: 10.1007/s00224-004-1155-5 (cited on page 97).

[AM20] P. K. Albers and G. McVean. "Dating Genomic Variants and Shared Ancestry in Population-Scale Sequencing Data". In: *Public Library of Science (PLoS) Biology* 18.1 (2020). Edited by N. H. Barton, e3000586. ISSN: 1545-7885. DOI: 10.1371/journal.pbio.3000586 (cited on page 15).

[APV18] A. B. Abecasis, M. Pingarilho, and A.-M. Vandamme. "Phylogenetic Analysis as a Forensic Tool in HIV Transmission Investigations". In: *AIDS* 32.5 (2018), pages 543–554. ISSN: 0269-9370. DOI: 10.1097/qad.0000000000001728 (cited on pages 1, 11).

[AS05] C. Ané and M. J. Sanderson. "Missing the Forest for the Trees: Phylogenetic Compression and Its Implications for Inferring Complex Evolutionary Histories". In: *Systematic Biology* 54.1 (2005). Edited by M. Steel, pages 146–157. DOI: 10.1080/10635150590905984 (cited on page 26).

[Ash13] J. Ashton. "Phylogenetic Methods in Drug Discovery". In: *Current Drug Discovery Technologies* 10.4 (2013), pages 255–262. ISSN: 1570-1638. DOI: 10.2174/15701638113109990033 (cited on page 11).

[ATD18] H. Atas, N. Tuncbag, and T. Doğan. "Phylogenetic and Other Conservation-Based Approaches to Predict Protein Functional Sites". In: *Computational Drug Discovery and Design*. Springer, 2018, pages 51–69. ISBN: 9781493977567. DOI: 10.1007/978-1-4939-7756-7_4 (cited on page 11).

[Aut+15] A. Auton, G. R. Abecasis, D. M. Altshuler, R. M. Durbin, G. R. Abecasis, et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (2015), pages 68–74. ISSN: 1476-4687. DOI: 10.1038/nature15393 (cited on pages 2, 32).

[Baa+19a] I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. "Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning". In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW '25)*. IEEE Press, 2019, pages 175–184. DOI: 10.1109/ipdpsw.2019.00038 (cited on page 123).

[Baa+19b] I. Baar, L. Hübner, P. Oettig, A. Zapletal, S. Schlag, A. Stamatakis, and B. Morel. *Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning*. Technical report. Karlsruhe Institute of Technology, 2019. DOI: 10.1101/579318 (cited on page 124).

[Bar+17] V. Bartsch, R. Machado, D. Merten, M. Rahn, and F.-J. Pfreundt. "GASPI/GPI In-Memory Checkpointing Library". In: *23rd European Conference on Parallel and Distributed Computing (Euro-Par)*. Edited by F. F. Rivera, T. F. Pena, and J. C. Cabaleiro. Volume 10417. Lecture Notes in Computer Science. Springer,

2017, pages 497–508. DOI: 10.1007/978-3-319-64203-1_36 (cited on pages 18, 42, 44, 45, 60).

[Bau+11] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems". In: *High Performance Computing Networking, Storage and Analysis (SC)*. Edited by S. A. Lathrop, J. Costa, and W. Kramer. ACM Press, 2011, 32:1–32:32. DOI: 10.1145/2063384.2063427 (cited on pages 18, 19, 42, 44, 46, 48).

[Bau15] H. Bauke. *MPL - A Message Passing Library*. 2015. URL: https://github.com/rabauke/mpl (cited on pages 85, 90).

[BdF24] J. J. Bernal-Gallardo and S. de Folter. "Plant Genome Information Facilitates Plant Functional Genomics". In: *Planta* 259.5 (2024). ISSN: 1432-2048. DOI: 10.1007/s00425-024-04397-z (cited on page 2).

[BDY16] B. Berger, N. M. Daniels, and Y. W. Yu. "Computational Biology in the 21st Century: Scaling with Compressive Algorithms". In: *Communincations of the ACM* 59.8 (2016), pages 72–80. ISSN: 1557-7317. DOI: 10.1145/2957324 (cited on page 1).

[BE12] C. G. Begley and L. M. Ellis. "Raise Standards for Preclinical Cancer Research". In: *Nature* 483.7391 (2012), pages 531–533. ISSN: 1476-4687. DOI: 10.1038/483531a (cited on page 19).

[Ben+12] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, et al. "GenBank". In: *Nucleic Acids Research* 41.D1 (2012), pages D36–D42. ISSN: 1362-4962. DOI: 10.1093/nar/gks1195 (cited on page 2).

[Ber+07] E. Bernard, Y. Azad, A. Vandamme, M. Weait, and A. Geretti. "HIV Forensics: Pitfalls and Acceptable Standards in the Use of Phylogenetic Analysis as Evidence In Criminal Investigations of HIV Transmission". In: *HIV Medicine* 8.6 (2007), pages 382–387. ISSN: 1468-1293. DOI: 10.1111/j.1468-1293.2007.00486.x (cited on pages 1, 11).

[BGK18] T. Bingmann, S. Gog, and F. Kurpicz. "Scalable Construction of Text Indexes with Thrill". In: *IEEE BigData*. IEEE Press, 2018, pages 634–643 (cited on page 102).

[BH14] M. Besta and T. Hoefler. "Fault Tolerance for Remote Memory Access Programming Models". In: *23rd High-Performance Parallel and Distributed Computing (HPDC)*. Edited by B. Plale, M. Ripeanu, F. Cappello, and D. Xu. ACM Press, 2014, pages 37–48. DOI: 10.1145/2600212.2600224 (cited on pages 18, 42, 44, 45, 48, 59).

[Bic+24] A. G. Bick, G. A. Metcalf, K. R. Mayo, L. Lichtenstein, S. Rura, R. J. Carroll, et al. "Genomic Data in the All of Us Research Program". In: *Nature* 627.8003 (2024), pages 340–346. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06957-x (cited on page 2).

[Bil+11] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. "Random Access to Grammar-Compressed Strings". In: *22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Edited by D. Randall. SIAM, 2011, pages 373–389. DOI: 10.1137/1.9781611973082.30 (cited on page 26).

[Bil+15] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. "Random Access to Grammar-Compressed Strings and Trees". In: *SIAM Journal on Computing (SICOMP)* 44.3 (2015), pages 513–539. DOI: 10.1137/130936889 (cited on page 26).

[Bin+16] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, et al. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *IEEE BigData*. IEEE Computer Society, 2016, pages 172–183 (cited on pages 108, 111).

[Bin18a] T. Bingmann. *pDCX*. 2018. URL: `https://github.com/bingmann/pDCX` (cited on page 101).

[Bin18b] T. Bingmann. "Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools". en. PhD thesis. Karlsruhe Institute of Technology, 2018. DOI: `10.5445/ir/1000085031` (cited on page 111).

[Bir+15] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, et al. "Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics". In: *23rd IEEE High-Performance Interconnects (HOTI)*. IEEE Press, 2015, pages 1–9. DOI: `10.1109/hoti.2015.22` (cited on page 15).

[BK13] P. Balaji and D. Kimpe. "On the Reproducibility of MPI Reduction Operations". In: *10th IEEE High Performance Computing and Communications (HPCC) & IEEE Embedded and Ubiquitous Computing (EUC)*. IEEE Press, 2013. DOI: `10.1109/hpcc.and.euc.2013.65` (cited on pages 64, 66–68, 70).

[Bla+13] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. "An Evaluation of User-Level Failure Mitigation Support in MPI". In: *Computing* 95.12 (2013), pages 1171–1184. DOI: `10.1007/s00607-013-0331-3` (cited on pages 19, 43, 51, 106).

[Bla+22] M. Blaxter, N. Mieszkowska, F. Di Palma, P. Holland, R. Durbin, T. Richards, et al. "Sequence locally, think globally: The Darwin Tree of Life Project". In: *Proceedings of the National Academy of Sciences* 119.4 (2022). ISSN: 1091-6490. DOI: `10.1073/pnas.2115642118` (cited on page 2).

[BMV01] D. A. Bader, B. M. Moret, and L. Vawter. "Industrial Applications of High-Performance Computing for Phylogeny Reconstruction". In: *Commercial Applications for High-Performance Computing*. Edited by H. J. Siegel. Volume 4528. Society of Photographic Instrumentation Engineers (SPIE), 2001, pages 159–168. DOI: `10.1117/12.434868` (cited on page 11).

[BN11] P. Bhattacharya and I. Neamtiu. "Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++". In: *33rd ACM International Conference on Software Engineering (ICSE)*. ICSE11. ACM Press, 2011. DOI: `10.1145/1985793.1985817` (cited on page 84).

[Boe15] C. Boettiger. "An Introduction to Docker for Reproducible Research". In: *ACM Special Interest Group on Operating Systems (SIGOPS)* 49.1 (2015), pages 71–79. DOI: `10.1145/2723872.2723882` (cited on page 20).

[Bor25] J. Borowitz. *Distributed Kernelization Techniques for the Maximum Weight Independent Set Problem*. en. Karlsruher Institut für Technologie (KIT), 2025. DOI: `10.5445/IR/1000179280` (cited on pages 107, 111).

[Bos+08] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. "Algorithmic Based Fault Tolerance Applied to High Performance Computing". In: *Journal of Parallel and Distributed Computing* (2008) (cited on page 17).

[Bou19] A. Bouteiller. *ULFM 4.0.2u1 Release Notes*. online. 2019 (cited on page 45).

[Bow18] C. Bowern. "Computational Phylogenetics". In: *Annual Review of Linguistics* 4.1 (2018), pages 281–296. ISSN: 2333-9691. DOI: `10.1146/annurev-linguistics-011516-034142` (cited on page 11).

[BP94] C. S. Baker and S. R. Palumbi. “Which Whales Are Hunted? A Molecular Genetic Approach to Monitoring Whaling”. In: *Science* 265.5178 (1994), pages 1538–1539. ISSN: 1095-9203. DOI: 10.1126/science.265.5178.1538 (cited on page 11).

[BR05] M. Bellare and P. Rogaway. *Intorudction to Modern Cryptography*. 2005 (cited on page 30).

[Bro+17] D. Brown, D. Smeets, B. Székely, D. Larsimont, A. M. Szász, P.-Y. Adnet, et al. “Phylogenetic Analysis of Metastatic Progression in Breast Cancer Using Somatic Mutations and Copy Number Aberrations”. In: *Nature Communications* 8.1 (2017). ISSN: 2041-1723. DOI: 10.1038/ncomms14944 (cited on page 11).

[Bro87] Brooks. “No Silver Bullet: Essence and Accidents of Software Engineering”. In: *Computer* 20.4 (1987), pages 10–19. ISSN: 0018-9162. DOI: 10.1109/mc.1987.1663532 (cited on page 3).

[Bry01] D. Bryant. “A Classification of Consensus Methods for Phylogenetics”. In: *Bioconsensus*. 2001 (cited on page 12).

[BS21] B. Bettisworth and A. Stamatakis. “Root Digger: A Root Placement Program for Phylogenetic Trees”. In: *BMC Bioinformatics* 22.1 (2021). ISSN: 1471-2105. DOI: 10.1186/s12859-021-03956-5 (cited on page 8).

[BSV03] R. Bhagwan, S. Savage, and G. M. Voelker. “Understanding Availability”. In: *Peer-to-Peer Systems II*. Springer, 2003, pages 256–267. ISBN: 9783540451723. DOI: 10.1007/978-3-540-45172-3_24 (cited on page 17).

[Bus+99] R. M. Bush, C. A. Bender, K. Subbarao, N. J. Cox, and W. M. Fitch. “Predicting the Evolution of Human Influenza A”. In: *Science* 286.5446 (1999), pages 1921–1925. ISSN: 1095-9203. DOI: 10.1126/science.286.5446.1921 (cited on page 11).

[BVG17] E. Birney, J. Vamathevan, and P. Goodhand. “Genomics in Healthcare: GA4GH looks to 2022”. In: *CoRR* (2017). DOI: 10.1101/203554 (cited on page 2).

[Byc+18] C. Bycroft, C. Freeman, D. Petkova, G. Band, L. T. Elliott, et al. “The UK Biobank Resource with Deep Phenotyping and Genomic Data”. In: *Nature* 562.7726 (2018), pages 203–209. ISSN: 1476-4687. DOI: 10.1038/s41586-018-0579-z (cited on page 2).

[Cap+14] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. “Toward Exascale Resilience: 2014 update”. In: *Supercomputing Frontiers and Innovations* 1.1 (2014), pages 5–28. DOI: 10.14529/jsfi140101 (cited on pages 3, 17, 42).

[Car+13] M. L. Carpenter, J. D. Buenrostro, C. Valdiosera, H. Schroeder, M. E. Allentoft, et al. “Pulling out the 1 %: Whole-Genome Capture for the Targeted Enrichment of Ancient DNA Sequencing Libraries”. In: *The American Journal of Human Genetics* 93.5 (2013), pages 852–864. ISSN: 0002-9297. DOI: 10.1016/j.ajhg.2013.10.002 (cited on pages 12, 24).

[Cas+12] E. Castro-Nallar, M. Pérez-Losada, G. F. Burton, and K. A. Crandall. “The Evolution of HIV: Inferences using Phylogenetics”. In: *Molecular Phylogenetics and Evolution* 62.2 (2012), pages 777–792. ISSN: 1055-7903. DOI: 10.1016/j.ympev.2011.11.019 (cited on pages 1, 11).

[CD96] T. Chiueh and P. Deng. “Evaluation of Checkpoint Mechanisms for Massively Parallel Machines”. In: *26th IEEE Fault-Tolerant Computing (FTCS)*. IEEE Computing Society, 1996, pages 370–379. DOI: 10.1109/ftcs.1996.534622 (cited on page 48).

[CG14] C. Colijn and J. Gardy. "Phylogenetic Tree Shapes Resolve Disease Transmission Patterns". In: *Evolution, Medicine, and Public Health* 2014.1 (2014), pages 96–108. ISSN: 2050-6201. DOI: 10.1093/emph/eou018 (cited on pages 1, 11).

[Che+18] S. Cheng, M. Melkonian, S. A. Smith, S. Brockington, J. M. Archibald, P.-M. Delaux, et al. "10KP: A Phylodiverse Genome Sequencing Plan". In: *GigaScience* 7.3 (2018). ISSN: 2047-217x. DOI: 10.1093/gigascience/giy013 (cited on page 2).

[CK18] M. J. Corden and D. Kreitzer. *Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?* Technical report. Software Services Group, Intel Corporation, 2018 (cited on pages 20, 21).

[Cle+13] M. A. Cleveland, T. A. Brunner, N. A. Gentile, and J. A. Keasler. "Obtaining Identical Results with Double Precision Global Accuracy on Different Numbers of Processors in Parallel Particle Monte Carlo Simulations". In: *Journal of Computational Physics* 251 (2013), pages 223–236. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2013.05.041 (cited on pages 3, 21).

[CLP16] C. Chohra, P. Langlois, and D. Parello. "Efficiency of Reproducible Level 1 BLAS". In: *Scientific Computing, Computer Arithmetic, and Validated Numerics*. Springer, 2016, pages 99–108. ISBN: 9783319317694. DOI: 10.1007/978-3-319-31769-4_8 (cited on pages 64, 68).

[CLP17] C. Chohra, P. Langlois, and D. Parello. "Reproducible, Accurately Rounded and Efficient BLAS". In: *23rd European Conference On Parallel and Distributed Computing (Euro-Par): Parallel Processing*. Springer, 2017, pages 609–620. ISBN: 9783319589435. DOI: 10.1007/978-3-319-58943-5_49 (cited on page 68).

[Col+15] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. "Numerical Reproducibility for the Parallel Reduction on Multi- and Many-core Architectures". In: *Parallel Computing* 49 (2015), pages 83–97. DOI: 10.1016/j.parco.2015.09.001 (cited on page 68).

[Coo+10] G. Coop, D. Witonsky, A. Di Rienzo, and J. K. Pritchard. "Using Environmental Correlations to Identify Loci Underlying Local Adaptation". In: *Genetics* 185.4 (2010), pages 1411–1423. ISSN: 1943-2631. DOI: 10.1534/genetics.110.114819 (cited on page 30).

[Cor+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2nd. MIT Press, 2001. ISBN: 0262032937 (cited on page 66).

[Cor18] A. Correa. *boost-mpi3*. 2018. URL: https://gitlab.com/correaa/boost-mpi3 (cited on page 90).

[CPW15] C. Collberg, T. Proebsting, and A. M. Warren. *Repeatability and Benefaction in Computer Systems Research*. Technical report. University of Arizona, 2015 (cited on page 20).

[Cra13] Cray. *Cray XK7 Specifications*. online. 2013 (cited on page 60).

[CS17] X. Chen and J. Sun. "Study and Analysis of the High Performance Computing Failures in China Meteorological Field". In: *Journal of Geoscience and Environment Protection* 05.12 (2017), pages 28–40. ISSN: 2327-4344. DOI: 10.4236/gep.2017.512002 (cited on page 17).

[CT05] B. Chor and T. Tuller. “Maximum Likelihood of Evolutionary Trees: Hardness and Approximation”. In: *Bioinformatics* 21.Suppl 1 (2005), pages i97–i106. ISSN: 1460-2059. DOI: 10.1093/bioinformatics/bti1027 (cited on pages 13, 14).

[Dal+08] L. Dalcín, R. Paz, M. A. Storti, and J. D’Elía. “MPI for Python: Performance Improvements and MPI-2 Extensions”. In: *Journal of Distributed Computing* 68.5 (2008), pages 655–662 (cited on pages 90, 91).

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859 (cited on pages 1, 10, 12, 24).

[Dar87] C. Darwin. *Charles Darwin’s Notebooks, 1836-1844*. Edited by P. G. Barrett, H. PJ, K. S, and S. D. British Museum of Natural History, 1987 (cited on pages 1, 10, 24).

[Dav+85] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. White. “The Evolution of Large-Scale Structure in a Universe Dominated by Cold Dark Matter”. In: *Astrophysical Journal* 292 (1985), pages 371–394 (cited on page 102).

[Dem+23] A. C. Demiralp, P. Martin, N. Sakic, M. Krüger, and T. Gerrits. “A C++20 Interface for MPI 4.0”. In: *CoRR* (2023) (cited on pages 85, 90).

[DF21] L. Dalcín and Y. L. Fang. “mpi4py: Status Update After 12 Years of Development”. In: *Computing in Science & Engineering* 23.4 (2021), pages 47–54 (cited on page 90).

[DFS18] D. Darriba, T. Flouri, and A. Stamatakis. “The State of Software for Evolutionary Biology”. In: *Molecular Biology and Evolution* 35.5 (2018), pages 1037–1046. ISSN: 0737-4038. DOI: 10.1093/molbev/msy014 (cited on pages 3, 20, 21, 38, 64, 67).

[DG04] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Osdi*. USENIX Association, 2004, pages 137–150 (cited on pages 108, 111).

[DHR15] J. Dongarra, T. Herault, and Y. Robert. “Fault Tolerance Techniques for High-Performance Computing”. In: T. Herault, Y. Robert, H. Casanova, F. Vivien, D. Zaidouni, and J. Dongarra. *Fault-Tolerance Techniques for High-Performance Computing*. Edited by T. Herault and Y. Robert. Springer, 2015. Chapter 1, pages 3–85. ISBN: 9783319209432. DOI: 10.1007/978-3-319-20943-2_1 (cited on pages 3, 17–19, 42).

[Di+19] S. Di, H. Guo, R. Gupta, E. Pershey, M. Snir, and F. Cappello. “Exploring Properties and Correlations of Fatal Events in a Large-Scale HPC System”. In: *IEEE Transactions on Parallel Distributed Systems* 30.2 (2019), pages 361–374. DOI: 10.1109/tpds.2018.2864184 (cited on page 17).

[Die12] K. Diethelm. “The Limits of Reproducibility in Numerical Simulation”. In: *Computing in Science & Engineering* 14.1 (2012), pages 64–72. ISSN: 1521-9615. DOI: 10.1109/mcse.2011.21 (cited on pages 3, 21, 38).

[DN13] J. Demmel and H. D. Nguyen. “Fast Reproducible Floating-Point Summation”. In: *21st IEEE Computer Arithmetic (ARITH)*. IEEE Press, 2013, pages 163–172 (cited on page 68).

[DN15] J. Demmel and H. D. Nguyen. “Parallel Reproducible Summation”. In: *IEEE Transactions on Computers* 64.7 (2015), pages 2060–2070. ISSN: 0018-9340. DOI: 10.1109/tc.2014.2345391 (cited on pages 3, 21, 64, 68).

[Dol18] R. Dolbeau. “Theoretical Peak FLOPS per Instruction Set: A Tutorial”. In: *The Journal of Supercomputing* 74.3 (2018), pages 1341–1377. ISSN: 0920-8542. DOI: 10.1007/s11227-017-2177-5 (cited on page 71).

[DPW24] D. DeHaas, Z. Pan, and X. Wei. “Enabling Efficient Analysis of Biobank-Scale Data with Genotype Representation Graphs”. In: *Nature Computational Science* 5.2 (2024), pages 112–124. ISSN: 2662-8457. DOI: 10.1038/s43588-024-00739-9 (cited on page 27).

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. “Variations on the Common Subexpression Problem”. In: *Journal of the ACM* 27.4 (1980), pages 758–771. ISSN: 1557-735x. DOI: 10.1145/322217.322228 (cited on page 26).

[Efr79] B. Efron. “Bootstrap Methods: Another Look at the Jackknife”. In: *The Annals of Statistics* 7.1 (1979). ISSN: 0090-5364. DOI: 10.1214/aos/1176344552 (cited on page 76).

[EG03] C. Engelmann and A. Geist. “A Diskless Checkpointing Algorithm for Super-Scale Architectures Applied to the Fast Fourier Transform”. In: *1st Challenges of Large Applications in Distributed Environments (CLADE)*. CLADE ’03. IEEE Computer Society, 2003, page 47. ISBN: 0769519849. DOI: 10.1109/clade.2003.1209999 (cited on pages 17, 19, 43).

[EHH96] B. Efron, E. Halloran, and S. Holmes. “Bootstrap Confidence Levels for Phylogenetic Trees”. In: *Proceedings of the National Academy of Sciences* 93.23 (1996), pages 13429–13429. ISSN: 1091-6490. DOI: 10.1073/pnas.93.23.13429 (cited on page 76).

[EK40] P. Erdos and M. Kac. “The Gaussian Law of Errors in the Theory of Additive Number Theoretic Functions”. In: *American Journal of Mathematics* 62.1/4 (1940), page 738. DOI: 10.2307/2371483 (cited on page 51).

[ES13] N. El-Sayed and B. Schroeder. “Reading Between the Lines of Failure Logs: Understanding How HPC Systems Fail”. In: *43rd IEEE/IFIP Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2013, pages 1–12. ISBN: 978-1-4673-6471-3. DOI: 10.1109/dsn.2013.6575356 (cited on page 17).

[EWV18] N. Eckshtain-Levi, A. J. Weisberg, and B. A. Vinatzer. “The Population Genetic Test Tajima’s D Identifies Genes Encoding Pathogen-Associated Molecular Patterns and Other Virulence-Related Genes in Ralstonia solanacearum”. In: *Molecular Plant Pathology* 19.9 (2018), pages 2187–2192. ISSN: 1364-3703. DOI: 10.1111/mpp.12688 (cited on page 15).

[Fel78] J. Felsenstein. “The Number of Evolutionary Trees”. In: *Systematic Zoology* 27.1 (1978), page 27. DOI: 10.2307/2412810 (cited on page 13).

[Fel81] J. Felsenstein. “Evolutionary Trees from DNA Sequences: A Maximum Likelihood Approach”. In: *Journal of Molecular Evolution* 17.6 (1981), pages 368–376. DOI: 10.1007/bf01734359 (cited on page 13).

[Fel85] J. Felsenstein. “Confidence Limits on Phylogenies: An Approach Using the Bootstrap”. In: *Evolution* 39.4 (1985), page 783. ISSN: 0014-3820. DOI: 10.2307/2408678 (cited on page 76).

[Fer17] J. L. Fernández-García. “Phylogenetics for Wildlife Conservation”. In: *Phylogenetics*. Edited by I. Y. Abdurakhmonov. Rijeka: InTech, 2017. Chapter 2. DOI: 10.5772/intechopen.69240 (cited on page 11).

[Fis31] R. A. Fisher. “The Distribution of Gene Ratios for Rare Mutations”. In: *Proceedings of the Royal Society of Edinburgh* 50 (1931), pages 204–219. ISSN: 0370-1646. DOI: 10.1017/s0370164600044886 (cited on page 14).

[FK19] J. Fischer and F. Kurpicz. “Lightweight Distributed Suffix Array Construction”. In: *Alenex*. SIAM, 2019, pages 27–38 (cited on page 102).

[Flo+14] T. Flouri, F. Izquierdo-Carrasco, D. Darriba, A. Aberer, L.-T. Nguyen, B. Minh, et al. “The Phylogenetic Likelihood Library”. In: *Systematic Biology* 64.2 (2014), pages 356–362. ISSN: 1063-5157. DOI: 10.1093/sysbio/syu084 (cited on page 8).

[Fox+80] G. Fox, E. Stackebrandt, R. Hespell, J. Gibson, J. Maniloff, T. Dyer, et al. “The Phylogeny of Prokaryotes”. In: *Science* 209.4455 (1980), pages 457–463. DOI: 10.1126/science.6771870 (cited on pages 1, 11).

[FS13] J. S. Forsberg and S. Soini. “A Big Step for Finnish Biobanking”. In: *Nature Reviews Genetics* 15.1 (2013), pages 6–6. ISSN: 1471-0064. DOI: 10.1038/nrg3646 (cited on page 2).

[Fun+19] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, et al. “Communication-Free Massively Distributed Graph Generation”. In: *Journal of Parallel Distributed Computing* 131 (2019), pages 200–217 (cited on page 106).

[FZ94] G. W. Furnas and J. Zacks. “Multitrees: Enriching and Reusing Hierarchical Structure”. In: *Conference on Human Factors in Computing Systems (CHI)*. Edited by B. Adelson, S. T. Dumais, and J. S. Olson. CHI ’94. 1994, pages 330–336. DOI: 10.1145/191666.191778 (cited on page 27).

[Gam+14] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. “Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales”. In: *High Performance Computing, Networking, Storage and Analysis (SC)*. Edited by T. Damkroger and J. J. Dongarra. IEEE Computer Society, 2014, pages 895–906. DOI: 10.1109/sc.2014.78 (cited on pages 17, 18, 42, 44, 45, 59).

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (cited on page 92).

[GG03] S. Guindon and O. Gascuel. “A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood”. In: *Systematic Biology* 52.5 (2003). Edited by B. Rannala, pages 696–704. DOI: 10.1080/10635150390235520 (cited on page 13).

[GH22] L. Gottesbüren and M. Hamann. “Deterministic Parallel Hypergraph Partitioning”. In: *28th European Conference on Parallel and Distributed Computing (Euro-Par): Parallel Processing*. Springer, 2022, pages 301–316. ISBN: 9783031125973. DOI: 10.1007/978-3-031-12597-3_19 (cited on page 20).

[Gho+21] S. Ghosh, C. Alsobrooks, M. Rüfenacht, A. Skjellum, P. V. Bangalore, and A. Lumsdaine. “Towards Modern C++ Language Support for MPI”. In: *Workshop on Extreme Scale MPI (ExaMPI)*. IEEE Press, 2021, pages 27–35 (cited on pages 85, 89, 90, 103).

[GJB13] L. M. Gattepaille, M. Jakobsson, and M. G. Blum. “Inferring Population Size Changes with Sequence and SNP Data: Lessons from Human Bottlenecks”. In: *Heredity* 110.5 (2013), pages 409–419. ISSN: 1365-2540. DOI: 10.1038/hdy.2012.120 (cited on page 15).

[GL19] S. Guttinger and A. C. Love. “Characterizing Scientific Failure: Putting the Replication Crisis in Context”. In: *EMBO reports* 20.9 (2019). ISSN: 1469-3178. DOI: 10.15252/embr.201948765 (cited on page 64).

[GL99] W. Gropp and E. L. Lusk. “Reproducible Measurements of MPI Performance Characteristics”. In: *European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Edited by J. J. Dongarra, E. Luque, and T. Margalef. Volume 1697. Lecture Notes in Computer Science. Springer, 1999, pages 11–18. ISBN: 3-540-66549-8. DOI: 10.1007/3-540-48158-3_2 (cited on page 20).

[GMM16] S. Goodwin, J. D. McPherson, and W. R. McCombie. “Coming of Age: Ten Years of Next-Generation Sequencing Technologies”. In: *Nature Reviews Genetics* 17.6 (2016), pages 333–351. ISSN: 1471-0064. DOI: 10.1038/nrg.2016.49 (cited on page 1).

[Gol91] D. Goldberg. “What Every Computer Scientist Should Know About Floating-Point Arithmetic”. In: *ACM Computing Society* 23.1 (1991), pages 5–48. ISSN: 1557-7341. DOI: 10.1145/103162.103163 (cited on pages 3, 20, 38).

[Gon14] F. González-Candelas. “La evolución en los tribunales: aplicaciones forenses de las filogenias moleculares”. In: *Metode Science Studies Journal* (2014), pages 107–113. ISSN: 2174-3487. DOI: 10.7203/metode.78.2451 (cited on page 11).

[Gos+19] T. I. Gossmann, A. Shanmugasundram, S. Börno, L. Duvaux, C. Lemaire, H. Kuhl, et al. “Ice-Age Climate Adaptations Trap the Alpine Marmot in a State of Low Genetic Diversity”. In: *Current Biology* 29.10 (2019), pages 1712–1720. DOI: 10.1016/j.cub.2019.04.020 (cited on page 10).

[Gos+21] M. J. Gossman, B. Nicolae, J. C. Calhoun, F. Cappello, and M. C. Smith. “Towards Aggregated Asynchronous Checkpointing”. In: *CoRR* (2021) (cited on page 44).

[Gra+06] R. Graham, G. Shipman, B. Barrett, R. Castain, G. Bosilca, and A. Lumsdaine. “Open MPI: A High-Performance, Heterogeneous MPI”. In: *IEEE Cluster Computing (CLUSTER) ’06*. IEEE Press, 2006, pages 1–9. DOI: 10.1109/clustr.2006.311904 (cited on page 16).

[Gre23] S. J. Greenhill. “Language Phylogenies: Modelling the Evolution of Language”. In: *The Oxford Handbook of Cultural Evolution*. Oxford University Press, 2023. ISBN: 9780191905780. DOI: 10.1093/oxfordhb/9780198869252.013.61 (cited on page 11).

[Gri+13] I. V. Grigoriev, R. Nikitin, S. Haridas, A. Kuo, R. Ohm, R. Otillar, et al. “MycoCosm Portal: Gearing up for 1000 Fungal Genomes”. In: *Nucleic Acids Research* 42.D1 (2013), pages D699–d704. ISSN: 1362-4962. DOI: 10.1093/nar/gkt1183 (cited on page 2).

[Gro02] W. Gropp. “MPICH2: A New Start for MPI Implementations”. In: *9th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg, 7: Springer, 2002 (cited on pages 19, 43).

[GS13] S. P. Gavaskar and C. D. V. Subbarao. “A Survey of Distributed Fault Tolerance Strategies”. In: *Journal of Advanced Research in Computer and Communication Engineering* 2.11 (2013). ISSN: 2278-1021 (cited on pages 19, 42).

[GT07] D. Gregor and M. Troyer. *Boost.MPI*. Version 1.84. 2007. URL: https://www.boost.org/doc/libs/1%5C%5F84%5C%5F0/doc/html/mpi.html (cited on pages 85, 89).

[Gup+17] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. "Failures in Large Scale Systems: Long-Term Measurement, Analysis, and Implications". In: *High Performance Computing, Networking, Storage and Analysis (SC)* (2017). DOI: 10.1145/3126908.3126937 (cited on page 17).

[GV17] W. S. Grant and R. Voorhies. *Cereal – A C++11 Library for Serialization*. 2017. URL: http://uscilab.github.io/cereal/ (cited on page 97).

[Haa+22a] J. Haag, D. Höhler, B. Bettisworth, and A. Stamatakis. "From Easy to Hopeless – Predicting the Difficulty of Phylogenetic Analyses". In: *Molecular Biology and Evolution* 39.12 (2022). Edited by N. Saitou. ISSN: 1537-1719. DOI: 10.1093/molbev/msac254 (cited on page 7).

[Haa+22b] J. Haag, L. Hübner, A. M. Kozlov, and A. Stamatakis. *The Free Lunch is not over yet – Systematic Exploration of Numerical Thresholds in Phylogenetic Inference*. Technical report. Heidelberg Institute for Theoretical Studies, 2022. DOI: 10.1101/2022.07.13.499893 (cited on page 124).

[Haa+23] J. Haag, L. Hübner, A. M. Kozlov, and A. Stamatakis. "The Free Lunch is not over yet – Systematic Exploration of Numerical Thresholds in Maximum Likelihood Phylogenetic Inference". In: *Bioinformatics Advances* 3.1 (2023). Edited by A. Ouangraoua. ISSN: 2635-0041. DOI: 10.1093/bioadv/vbad124 (cited on pages 7, 123).

[Haa+24] M. Haag, F. Kurpicz, P. Sanders, and M. Schimek. *Fast and Lightweight Distributed Suffix Array Construction – First Results*. 2024. DOI: 10.48550/ARXIV.2412.10160 (cited on pages 107, 111).

[Haa21] J. Haag. "Empirical Numerical Properties of Maximum Likelihood Phylogenetic Inference". Master's Thesis. Karlsruhe Institute of Technology, 2021 (cited on pages 7, 125).

[Hae66] E. Haeckel. *Allgemeine Anatomie der Organismen*. Georg Reimer, 1866 (cited on pages 1, 10, 12).

[HB05] D. H. Huson and D. Bryant. "Application of Phylogenetic Networks in Evolutionary Studies". In: *Molecular Biology and Evolution* 23.2 (2005), pages 254–267. ISSN: 0737-4038. DOI: 10.1093/molbev/msj030 (cited on page 12).

[HD01] Y. He and C. H. Q. Ding. "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications". In: *The Journal of Supercomputing* 18.3 (2001), pages 259–277. ISSN: 0920-8542. DOI: 10.1023/a:1008153532043 (cited on pages 3, 21, 68).

[HD06] P. H. Hargrove and J. C. Duell. "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters". In: *Scientific Discovery though Advanced Computing (SciDAC)*. Volume 46. IOP Publishing, 2006, pages 494–499. DOI: 10.1088/1742-6596/46/1/067 (cited on page 18).

[HD62] T. E. Hull and A. R. Dobell. "Random Number Generators". In: *(SIAM) Review* 4.3 (1962), pages 230–254. DOI: 10.1137/1004061 (cited on page 51).

[Hed92] S. B. Hedges. "The Number of Replications Needed for Accurate Estimation of the Bootstrap p Value in Phylogenetic Studies". In: *Molecular Biology and Evolution* (1992). ISSN: 1537-1719. DOI: 10.1093/oxfordjournals.molbev.a040725 (cited on page 76).

[Hen24] J. Hengstler. "Re-Engineering Genomic Tree Sequence Inference Algorithms". Master's Thesis. Karlsruhe Institute of Technology, 2024 (cited on pages 6, 38, 125).

[Her+15] T. Herault, Y. Robert, H. Casanova, F. Vivien, D. Zaidouni, and J. Dongarra. *Fault-Tolerance Techniques for High-Performance Computing*. Edited by T. Herault and Y. Robert. Springer, 2015. ISBN: 9783319209432. DOI: 10.1007/978-3-319-20943-2 (cited on page 42).

[Hér+19] T. Hérault, Y. Robert, A. Bouteiller, D. C. Arnold, K. B. Ferreira, G. Bosilca, et al. "Checkpointing Strategies for Shared High-Performance Computing Platforms". In: *Journal of Networking and Computing* 9.1 (2019), pages 28–52. DOI: 10.15803/ijnc.9.1_28 (cited on page 44).

[Hes+23] D. Hespe, L. Hübner, L. Hübschle-Schneider, P. Sanders, and D. Schreiber. "Scalable Discrete Algorithms for Big Data Applications". In: *High Performance Computing in Science and Engineering '21*. Springer, 2023, pages 439–449. ISBN: 9783031179372. DOI: 10.1007/978-3-031-17937-2_27 (cited on page 123).

[Hes+24a] D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, and T. N. Uhl. "Brief Announcement: (Near) Zero-Overhead C++ Bindings for MPI". In: *36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Volume 31. SPAA '24. ACM Press, 2024, pages 289–291. DOI: 10.1145/3626183.3660260 (cited on page 123).

[Hes+24b] D. Hespe, L. Hübner, C. Mercatoris, and P. Sanders. *Scalable Fault-Tolerant MapReduce*. Technical report. Karlsruhe Institute of Technology, 2024. DOI: 10.48550/arxiv.2411.16255 (cited on pages 8, 124).

[Hes23] L. D. Hespe. "Enabling Scalability: Graph Hierarchies and Fault Tolerance". en. PhD thesis. Karlsruhe Institute of Technology, 2023. DOI: 10.5445/ir/1000161317 (cited on page 111).

[HFR20] P. A. Hohenlohe, W. C. Funk, and O. P. Rajora. "Population Genomics for Wildlife Conservation and Management". In: *Molecular Ecology* 30.1 (2020), pages 62–82. ISSN: 1365-294x. DOI: 10.1111/mec.15720 (cited on pages 1, 12, 24).

[Hig93] N. J. Higham. "The Accuracy of Floating Point Summation". In: *SIAM Journal on Scientific Computing* 14.4 (1993), pages 783–799. ISSN: 1095-7197. DOI: 10.1137/0914050 (cited on pages 70, 80).

[Hin+15] C. E. Hinchliff, S. A. Smith, J. F. Allman, J. G. Burleigh, R. Chaudhary, et al. "Synthesis of Phylogeny and Taxonomy into a Comprehensive Tree of Life". In: *Proceedings of the National Academy of Sciences* 112.41 (2015), pages 12764–12769. ISSN: 1091-6490. DOI: 10.1073/pnas.1423041112 (cited on pages 1, 10, 12, 24).

[Hit40] E. Hitchcock. *Elementary Geology*. 1840 (cited on pages 1, 10, 12).

[HM03] B. Holland and V. Moulton. "Consensus Networks: A Method for Visualising Incompatibilities in Collections of Trees". In: *Algorithms in Bioinformatics*. 2003, pages 165–176. ISBN: 9783540397632. DOI: 10.1007/978-3-540-39763-2_13 (cited on page 12).

[Höt+14] J. Hötzer, M. Jainta, A. Vondrous, J. Ettrich, A. August, D. Stubenvoll, et al. "Phase-Field Simulations of Large-Scale Microstructures by Integrated Parallel Algorithms". In: *High Performance Computing in Science and Engineering*. Springer. 2014, pages 629–644 (cited on page 102).

[HRC09] T. J. Hacker, F. Romero, and C. D. Carothers. "An Analysis of Clustered Failures on Large Supercomputing Systems". In: *Journal of Parallel and Distributed Computing* 69.7 (2009), pages 652–665. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2009.03.007 (cited on page 17).

[HS24a] L. Hübner and A. Stamatakis. *Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests*. Technical report. Karlsruhe Institute of Technology, 2024. DOI: 10.1101/2024.05.23.595533 (cited on page 124).

[HS24b] L. Hübner and A. Stamatakis. "Memoization on Shared Subtrees Accelerates Computations on Genealogical Forests". In: *24th Workshop on Algorithms in Bioinformatics (WABI)*. Edited by S. P. Pissis and W. Sung. Volume 312. Leibniz International Proceedings in Informatics (LIPcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pages 1–22. ISBN: 978-3-95977-340-9. DOI: 10.4230/lipics.wabi.2024.5 (cited on pages 9, 23, 123).

[HSL10] T. Hoefler, C. Siebert, and A. Lumsdaine. "Scalable Communication Protocols for Dynamic Sparse Data Exchange". In: *Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, 2010, pages 159–168 (cited on page 106).

[Hua+12] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. "Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity". In: *ACM Special Interests Group on Programming Languages (SIGPLAN) Notices* 47.10 (2012), pages 879–896. ISSN: 1558-1160. DOI: 10.1145/2398857.2384680 (cited on page 111).

[Hüb+21a] L. Hübner, A. M. Kozlov, D. Hespe, P. Sanders, and A. Stamatakis. *Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG*. Technical report. Karlsruhe Institute of Technology, 2021. DOI: 10.1101/2021.01.15.426773 (cited on page 124).

[Hüb+21b] L. Hübner, A. M. Kozlov, D. Hespe, P. Sanders, and A. Stamatakis. "Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". In: *Bioinformatics* 37.22 (2021). Edited by R. Schwartz, pages 4056–4063. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab399 (cited on pages 9, 17–19, 42, 43, 49, 52, 57, 61, 110, 123).

[Hüb+22a] L. Hübner, D. Hespe, P. Sanders, and A. Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *12th IEEE/ACM Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE Press, 2022, pages 24–35. DOI: 10.1109/ftxs56515.2022.00008 (cited on pages 9, 41, 45, 123).

[Hüb+22b] L. Hübner, D. Hespe, P. Sanders, and A. Stamatakis. *ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms*. Technical report. Karlsruhe Institute of Technology, 2022. DOI: 10.48550/arxiv.2203.01107 (cited on pages 41, 124).

[Hüb14] L. Hübner. "Coloring Complex Networks". Bachelor's Thesis. Karlsruhe Institute of Technology, 2014 (cited on page 124).

[Hüb17] L. Hübner. "Analysis of Epigenetic Modulators Across Multiple Cancer Types". Bachelor's Thesis. Heidelberg University, 2017 (cited on page 124).

[Hüb20] L. Hübner. "Load-Balance and Fault-Tolerance for Massively Parallel Phylogenetic Inference". Master's thesis. Karlsruhe Institute of Technology, 2020 (cited on page 124).

[Hud83] R. R. Hudson. "Properties of a Neutral Allele Model with Intragenic Recombination". In: *Theoretical Population Biology* 23.2 (1983), pages 183–201. ISSN: 0040-5809. DOI: 10.1016/0040-5809(83)90013-8 (cited on pages 1, 11, 12, 25).

[Hug+23] L. J. Hughes, M. R. Massam, O. Morton, F. A. Edwards, B. R. Scheffers, and D. P. Edwards. "Global Hotspots of Traded Phylogenetic and Functional Diversity". In: *Nature* 620.7973 (2023), pages 351–357. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06371-3 (cited on pages 1, 11).

[HW09] K. E. Holsinger and B. S. Weir. "Genetics in Geographically Structured Populations: Defining, Estimating and Interpreting FST". In: *Nature Reviews Genetics* 10.9 (2009), pages 639–650. ISSN: 1471-0064. DOI: 10.1038/nrg2611 (cited on page 15).

[HW60] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Clarendon Press, 1960 (cited on page 51).

[HZ15] F. V. Henri Casanova and D. Zaidouni. "Using Replication for Resilience on Exascale Systems". In: T. Herault, Y. Robert, H. Casanova, F. Vivien, D. Zaidouni, and J. Dongarra. *Fault-Tolerance Techniques for High-Performance Computing*. Edited by T. Herault and Y. Robert. Springer, 2015. Chapter 4, pages 3–85. ISBN: 9783319209432. DOI: 10.1007/978-3-319-20943-2_4 (cited on page 17).

[IA20] F. Ingels and R. Azais. "A Reverse Search Method for the Enumeration of Unordered Forests using DAG Compression". In: *4th Workshop on Enumeration Problems and Applications (WEPA)* (2020) (cited on page 26).

[Iak+15] R. Iakymchuk, C. Collange, D. Defour, and S. Graillat. "ExBLAS: Reproducible and accurate BLAS library". In: *Numerical Reproducibility at Exascale*. 2015 (cited on page 68).

[IEEE754] *754-2019 - IEEE Standard for Floating-Point Arithmetic*. 2019. DOI: 10.1109/ieeestd.2019.8766229 (cited on pages 3, 20, 38, 106).

[IETF22] Internet Engineering Task Force. *Transmission Control Protocol*. Request for Comments 9293. MTI Systems, 2022 (cited on page 15).

[Ing22] F. Ingels. "On the Similarities of Trees: The Interest of Enumeration and Compression Methods. (Sur la similarité des arbres: l'intérêt des méthodes d'énumération et de compression)". PhD thesis. École normale supérieure de Lyon, France, 2022 (cited on page 26).

[Int24] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel. 2024 (cited on pages 20, 21).

[Int25] Intel. *Intel MPI Library*. 2025. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html (cited on page 16).

[Int25a] Intel oneMKL. *Floating Point Conditional Numerical Reproducibility on CPU and GPU*. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-the-conditional-numerical-reproducibility-cnr.html (cited on page 20).

[Int25b] Intel oneAPI. *Intel Compiler Developer Guide and Reference: arch-consistency*. 2025. URL: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2025-0/fimf-arch-consistency-qimf-arch-consistency.html (cited on page 20).

[Int25c] Intel MPI Library. *Developer Reference for Linux OS: I_MPI_ADJUST Familiy Environment Variables*. 2025. URL: https://cdrdv2.intel.com/v1/dl/getContent/835542 (cited on page 68).

[IT18] P. Ivie and D. Thain. "Reproducibility in Scientific Computing". In: *ACM Computing Surveys* 51.3 (2018), pages 1–36. ISSN: 1557-7341. DOI: 10.1145/3186266 (cited on pages 3, 19, 20, 64, 106).

[Jac89] G. Jacobson. "Space-Efficient Static Trees and Graphs". In: *30th IEEE Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1989, pages 549–554. DOI: 10.1109/sfcs.1989.63533 (cited on pages 27, 31).

[Jäg18] G. Jäger. "Global-Scale Phylogenetic Linguistic Inference from Lexical Resources". In: *Scientific Data* 5.1 (2018). ISSN: 2052-4463. DOI: 10.1038/sdata.2018.189 (cited on page 11).

[JAM20] S. F. Jalal Apostal, D. Apostal, and R. Marsh. "Improving Numerical Reproducibility of Scientific Software in Parallel Systems". In: *IEEE Electro Information Technology (EIT)*. IEEE Press, 2020, pages 066–074. DOI: 10.1109/eit48999.2020.9208338 (cited on page 64).

[Jar+14] E. D. Jarvis, S. Mirarab, A. J. Aberer, B. Li, P. Houde, C. Li, et al. "Whole-Genome Analyses Resolve Early Branches in the Tree of Life of Modern Birds". In: *Science* 346.6215 (2014), pages 1320–1331. DOI: 10.1126/science.1253451 (cited on page 18).

[Kah65] W. Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". In: *Communications of the ACM* 8.1 (1965), page 40 (cited on page 68).

[Kar+20] K. J. Karczewski, L. C. Francioli, G. Tiao, B. B. Cummings, J. Alföldi, et al. "The Mutational Constraint Spectrum Quantified from Variation in 141,456 Humans". In: *Nature* 581.7809 (2020), pages 434–443. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2308-7 (cited on page 2).

[Kel+18] J. Kelleher, K. R. Thornton, J. Ashander, and P. L. Ralph. "Efficient Pedigree Recording for Fast Population Genetics Simulation". In: *Public Library of Science (PLoS) Computational Biology* 14.11 (2018). Edited by S. L. Kosakovsky Pond, e1006581. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1006581 (cited on page 14).

[Kel+19] J. Kelleher, Y. Wong, A. W. Wohns, C. Fadil, P. K. Albers, and G. McVean. "Inferring Whole-Genome Histories in Large Population Datasets". In: *Nature Genetics* 51.9 (2019), pages 1330–1338. ISSN: 1546-1718. DOI: 10.1038/s41588-019-0483-y (cited on pages 11, 14, 25, 26, 32, 38).

[KEM16] J. Kelleher, A. M. Etheridge, and G. McVean. "Efficient Coalescent Simulation and Genealogical Analysis for Large Sample Sizes". In: *Public Library of Science (PLoS) Computational Biology* 12.5 (2016). Edited by Y. S. Song, e1004842. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1004842 (cited on pages 25, 26).

[KF13] U. Kang and C. Faloutsos. "Big Graph Mining: Algorithms and Discoveries". In: *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter* 14.2 (2013), pages 29–36. ISSN: 1931-0153. DOI: 10.1145/2481244.2481249 (cited on page 102).

[KF94] M. K. Kuhner and J. Felsenstein. "A Simulation Comparison of Phylogeny Algorithms Under Equal and Unequal Evolutionary Rates." In: *Molecular Biology and Evolution* (1994). DOI: 10.1093/oxfordjournals.molbev.a040126 (cited on page 13).

[KKV03] L. V. Kalé, S. Kumar, and K. Varadarajan. "A Framework for Collective Personalized Communication". In: *17th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2003, page 69. DOI: 10.1109/ipdps.2003.1213166 (cited on page 105).

[KL05] M. Kowaluk and A. Lingas. “LCA Queries in Directed Acyclic Graphs”. In: *32nd International Colloquium on Automata, Languages and Programming (ICALP)*. Edited by L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung. Volume 3580. Lecture Notes in Computer Science. Springer, 2005, pages 241–248. DOI: 10.1007/11523468_{2}{0} (cited on page 29).

[Knü+12] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, et al. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”. In: *Tools for High Performance Computing*. Springer, 2012, pages 79–91. ISBN: 9783642314766. DOI: 10.1007/978-3-642-31476-6_7 (cited on pages 76, 77, 79).

[Koh+17] N. Kohl, J. Hötzer, F. Schornbaum, M. Bauer, C. Godenschwager, H. Köstler, et al. “A Scalable and Extensible Checkpointing Scheme for Massively Parallel Simulations”. In: *High Performance Computing Applications* (2017) (cited on pages 17, 19).

[Kor00] B. Korber. “Timing the Ancestor of the HIV-1 Pandemic Strains”. In: *Science* 288.5472 (2000), pages 1789–1796. DOI: 10.1126/science.288.5472.1789 (cited on page 11).

[Koz+19] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis. “RAxML-NG: A Fast, Scalable and User-Friendly Tool for Maximum Likelihood Phylogenetic Inference”. In: *Bioinformatics* 35.21 (2019), pages 4453–4455. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz305 (cited on pages 13, 49, 57, 61, 65, 74, 86, 104, 110).

[Koz18] A. Kozlov. “Models, Optimizations, and Tools for Large-Scale Phylogenetic Inference, Handling Sequence Uncertainty, and Taxonomic Validation”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2018 (cited on page 49).

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. “Linear Work Suffix Array Construction”. In: *Journal of the ACM* 53.6 (2006), pages 918–936 (cited on page 101).

[Kur+24] F. Kurpicz, P. Mehnert, P. Sanders, and M. Schimek. “Scalable Distributed String Sorting”. en. In: *32nd European Symposium on Algorithms (ESA)*. 2024. DOI: 10.4230/LIPICS.ESA.2024.83 (cited on pages 107, 111).

[Lag+16] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, et al. “Evaluating and Extending User-Level Fault Tolerance in MPI Applications”. In: *Journal of High Performance Computing Applications* 30.3 (2016), pages 305–319. DOI: 10.1177/1094342015623623 (cited on pages 19, 43).

[Lag+19] I. Laguna, R. J. Marshall, K. M. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana. “A Large-Scale Study of MPI Usage in Open-Source HPC Applications”. In: *High Performance Computing, Networking, Storage, and Analysis (SC)*. ACM Press, 2019, 31:1–31:14 (cited on pages 84, 86, 107, 111).

[Lar13] B. Larget. “The Estimation of Tree Posterior Probabilities Using Conditional Clade Probability Distributions”. In: *Systematic Biology* 62.4 (2013), pages 501–511. DOI: 10.1093/sysbio/syt014 (cited on page 27).

[Las+09] O. Lascu, K. Bosworth, H. Kithöfer, J. Ko, N. Mroz, G. Noble, et al. *HPC Clusters Using InfiniBand on IBM Power Systems Servers*. IBM, 2009. ISBN: 9780738433394 (cited on page 15).

[Lea+15] A. D. Leaché, B. L. Banbury, J. Felsenstein, A. n.-M. de Oca, and A. Stamatakis. "Short Tree, Long Tree, Right Tree, Wrong Tree: New Acquisition Bias Corrections for Inferring SNP Phylogenies". In: *Systematic Biology* 64.6 (2015), pages 1032–1047. ISSN: 1076-836x. DOI: 10.1093/sysbio/syv053 (cited on page 13).

[Lei+14] L. Leitsalu, T. Haller, T. Esko, M.-L. Tammesoo, H. Alavere, H. Snieder, et al. "Cohort Profile: Estonian Biobank of the Estonian Genome Center". In: *Journal of Epidemiology* 44.4 (2014), pages 1137–1147. ISSN: 1464-3685. DOI: 10.1093/ije/dyt268 (cited on page 2).

[Lei+23] X. Lei, T. Gu, S. Graillat, X. Xu, and J. Meng. "Comparison of Reproducible Parallel Preconditioned BiCGSTAB Algorithm Based on ExBLAS and ReproBLAS". In: *High Performance Computing in Asia-Pacific Region*. HPC-Asia 2023. ACM Press, 2023, pages 46–54. DOI: 10.1145/3578178.3578234 (cited on page 68).

[Lew+18] H. A. Lewin, G. E. Robinson, W. J. Kress, W. J. Baker, J. Coddington, et al. "Earth BioGenome Project: Sequencing Life for the Future of Life". In: *Proceedings of the National Academy of Sciences* 115.17 (2018), pages 4325–4333. ISSN: 1091-6490. DOI: 10.1073/pnas.1720115115 (cited on page 2).

[LH66] R. C. Lewontin and J. L. Hubby. "A Molecular Approach to the Study of Genic Heterozygosity in Natural Populations. II. Amount of Variation and Degree of Heterozygosity in Natural Populations of Drosophila pseudoobscura". In: *Genetics* 54.2 (1966), pages 595–609. ISSN: 1943-2631. DOI: 10.1093/genetics/54.2.595 (cited on pages 12, 24).

[Li+19] H. Li, K. Wu, C. Ruan, J. Pan, Y. Wang, and H. Long. "Cost-Reduction Strategies in Massive Genomics Experiments". In: *Marine Life Science & Technology* 1.1 (2019), pages 15–21. ISSN: 2662-1746. DOI: 10.1007/s42995-019-00013-2 (cited on page 1).

[Li+23] K. Li, K. He, S. Graillat, H. Jiang, T. Gu, and J. Liu. "Multi-Level Parallel Multi-Layer Block Reproducible Summation Algorithm". In: *Parallel Computing* 115 (2023), page 102996. ISSN: 0167-8191. DOI: 10.1016/j.parco.2023.102996 (cited on pages 3, 21, 66, 68).

[Lia+06] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. K. Sahoo. "BlueGene/L Failure Analysis and Prediction Models". In: *46th IEEE/IFIP Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2006, pages 425–434. ISBN: 0-7695-2607-1. DOI: 10.1109/dsn.2006.18 (cited on page 17).

[Lip+22] M. Lipson, E. A. Sawchuk, J. C. Thompson, J. Oppenheimer, C. A. Tryon, et al. "Ancient DNA and Deep Population Structure in Sub-Saharan African Foragers". In: *Nature* 603.7900 (2022), pages 290–296. ISSN: 1476-4687. DOI: 10.1038/s41586-022-04430-9 (cited on pages 12, 24).

[LSS10] R. Leinonen, H. Sugawara, and M. Shumway. "The Sequence Read Archive". In: *Nucleic Acids Research* 39.Database (2010), pages D19–d21. ISSN: 1362-4962. DOI: 10.1093/nar/gkq1019 (cited on page 2).

[Lu+24] H. J. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell, editors. *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models). Version 1.0*. 2024 (cited on page 21).

[Lu05] C.-D. Lu. “Scalable Diskless Checkpointing for Large Parallel Systems”. PhD thesis. University of Illinois at Urbana-Champaign, 2005 (cited on pages 18, 42, 44, 45, 48, 60).

[Lu13] C.-D. Lu. “Failure Data Analysis of HPC Systems”. In: *Computer Science* (2013) (cited on page 17).

[Lui+18] G. Luikart, M. Kardos, B. K. Hand, O. P. Rajora, S. N. Aitken, et al. “Population Genomics: Advancing Understanding of Nature”. In: *Population Genomics*. Springer, 2018, pages 3–79. ISBN: 9783030045890. DOI: 10.1007/13836_2018_60 (cited on pages 1, 12, 24).

[Mac67] J. MacQueen. “Some Methods for Classification and Analysis of Multivariate Observations”. In: *5th Berkeley Symposium on Mathematical Statistics and Probability*. Volume 1. Oakland, CA, USA. 1967, pages 281–297 (cited on page 56).

[Mal+16] S. Mallick, H. Li, M. Lipson, I. Mathieson, M. Gymrek, F. Racimo, et al. “The Simons Genome Diversity Project: 300 Genomes from 142 Diverse Populations”. In: *Nature* 538.7624 (2016), pages 201–206. ISSN: 1476-4687. DOI: 10.1038/nature18964 (cited on pages 2, 32).

[Mar+14] C. D. Martino, Z. T. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. “Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters”. In: *44th IEEE/IFIP Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2014, pages 610–621. ISBN: 978-1-4799-2233-8. DOI: 10.1109/dsn.2014.62 (cited on page 17).

[Mar+15] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. K. Iyer. “Measuring and Understanding Extreme-Scale Application Resilience: A Field Study of 5, 000, 000 HPC Application Runs”. In: *45th IEEE/IFIP Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2015, pages 25–36. ISBN: 978-1-4799-8629-3. DOI: 10.1109/dsn.2015.50 (cited on page 17).

[Mem+16] B. Memishi, S. Ibrahim, M. S. Pérez, and G. Antoniu. “Fault Tolerance in MapReduce: A Survey”. In: *Resource Management for Big Data Platforms - Algorithms, Modelling, and High-Performance Computing Techniques*. Edited by F. Pop, J. Kolodziej, and B. D. Martino. Computer Communications and Networks. Springer, 2016, pages 205–240. ISBN: 978-3-319-44881-7. DOI: 10.1007/978-3-319-44881-7_11 (cited on pages 17, 111).

[Mer21] C. Mercatoris. “Scalable Decentralized Fault-Toleranc MapReduce for Iterative Algorithms”. Master’s Thesis. Karlsruhe Institute of Technology, 2021 (cited on page 125).

[Mes10] J. P. Mesirov. “Accessible Reproducible Research”. In: *Science* 327.5964 (2010), pages 415–416. ISSN: 1095-9203. DOI: 10.1126/science.1179653 (cited on pages 3, 19, 64).

[MHS19] G. Munjal, M. Hanmandlu, and S. Srivastava. “Phylogenetics Algorithms and Applications”. In: *Ambient Communications and Computer Systems*. Springer, 2019, pages 187–194. ISBN: 9789811359347. DOI: 10.1007/978-981-13-5934-7_17 (cited on page 11).

[Mic22] Microsoft. *x64 Calling Convention*. 2022. URL: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention (cited on page 21).

[Mil+08] C. D. Millar, L. Huynen, S. Subramanian, E. Mohandesan, and D. M. Lambert. “New Developments in Ancient Genomics”. In: *Trends in Evology & Evolution*

23.7 (2008), pages 386–393. ISSN: 0169-5347. DOI: 10.1016/j.tree.2008.04.002 (cited on pages 12, 24).

[Mil78] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pages 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4 (cited on page 88).

[Mis+14] B. Misof, S. Liu, K. Meusemann, R. S. Peters, A. Donath, C. Mayer, et al. "Phylogenomics Resolves the Timing and Pattern of Insect Evolution". In: *Science* 346.6210 (2014), pages 763–767. DOI: 10.1126/science.1257570 (cited on page 18).

[MM93] U. Manber and E. W. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM Journal on Computing (SICOMP)* 22.5 (1993), pages 935–948 (cited on pages 86, 101).

[Mon+13] C. L. Monteil, R. Cai, H. Liu, M. E. Mechan Llontop, S. Leman, D. J. Studholme, et al. "Nonagricultural Reservoirs Contribute to Emergence and Evolution of Pseudomonas Syringae Crop Pathogens". In: *New Phytologist* 199.3 (2013), pages 800–811. ISSN: 1469-8137. DOI: 10.1111/nph.12316 (cited on page 15).

[Moo+10] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System". In: *High Performance Computing Networking, Storage and Analysis (SC)*. IEEE Press, 2010, pages 1–11. DOI: 10.1109/sc.2010.18 (cited on pages 18, 42, 44, 45).

[Moo75] G. Moore. "Progress in Digital Integrated Electronics". In: *Technical Digest. International Electron Devices Meeting*. 1975 (cited on page 1).

[Mor+20a] B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mamais, M. A. Kozlov, P. Pavlidis, D. Paraskevis, and A. Stamatakis. "Phylogenetic Analysis of SARS-CoV-2 Data Is Difficult". In: *Molecular Biology and Evolution* 38.5 (2020). Edited by H. Malik, pages 1777–1791. ISSN: 1537-1719. DOI: 10.1093/molbev/msaa314 (cited on pages 7, 11, 123).

[Mor+20b] B. Morel, P. Barbera, L. Czech, B. Bettisworth, L. Hübner, S. Lutteropp, D. Serdari, E.-G. Kostaki, I. Mmai, M. A. Kozlov, P. Pavlidis, D. Paraskevis, and A. Stamatakis. *Phylogenetic Analysis of SARS-CoV-2 Data is Difficult*. Technical report. Heidelberg Institute for Theoretical Studies, 2020. DOI: 10.1101/2020.08.05.239046 (cited on page 124).

[Mor+20c] B. Morel, A. M. Kozlov, A. Stamatakis, and G. J. Szöllősi. "GeneRax: A Tool for Species-Tree-Aware Maximum Likelihood-Based Gene Family Tree Inference under Gene Duplication, Transfer, and Loss". In: *Molecular Biology and Evolution* 37.9 (2020). Edited by R. Nielsen, pages 2763–2774. ISSN: 1537-1719. DOI: 10.1093/molbev/msaa141 (cited on page 8).

[Mor+24] B. Morel, T. A. Williams, A. Stamatakis, and G. J. Szöllősi. "AleRax: A Tool for Gene and Species Tree Co-Estimation and Reconciliation Under a Probabilistic Model of Gene Duplication, Transfer, and Loss". In: *Bioinformatics* 40.4 (2024). Edited by R. Schwartz. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btae162 (cited on page 8).

[Mor16] D. A. Morrison. "Genealogies: Pedigrees and Phylogenies are Reticulating Networks Not Just Divergent Trees". In: *Evolutionary Biology* 43.4 (2016), pages 456–473. ISSN: 1934-2845. DOI: 10.1007/s11692-016-9376-5 (cited on pages 1, 11, 25).

[MPI2.2] *MPI 2.2: A Message-Passing Interface Standard*. MPI Forum, 2009 (cited on page 84).

[MPI20] MPI Forum. *What Features Do Users Need from an MPI C++ Interface?* GitHub issue. 2020. URL: https://github.com/mpi-forum/mpi-issues/issues/288 (cited on pages 85, 89).

[MPI4.1] *MPI 4.1: A Message-Passing Interface Standard*. MPI Forum, 2023 (cited on pages 3, 15, 16, 68, 69, 84, 86–88, 91, 96, 98).

[MPI5.0] *MPI 5.0: A Message-Passing Interface Standard (Draft)*. MPI Forum, 2025 (cited on page 89).

[MPICH] MPICH. *MPICH*. 2025. URL: https://www.mpich.org/ (cited on page 16).

[MSW10] S. J. Matthews, S. Sul, and T. L. Williams. "A Novel Approach for Compressing Phylogenetic Trees". In: *6th International Symposium on Bioinformatics Research and Applications (ISBRA)*. Edited by M. Borodovsky, J. P. Gogarten, T. M. Przytycka, and S. Rajasekaran. Volume 6053. Lecture Notes in Computer Science. Springer, 2010, pages 113–124. DOI: 10.1007/978-3-642-13078-6_{1}{3} (cited on page 26).

[MWS22] B. Morel, T. A. Williams, and A. Stamatakis. "Asteroid: A New Algorithm to Infer Species Trees from Gene Trees Under High Proportions of Missing Data". In: *Bioinformatics* 39.1 (2022). Edited by R. Schwartz. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btac832 (cited on page 8).

[Nea15] R. M. Neal. "Fast Exact Summation Using Small and Large Superaccumulators". In: *CoRR* (2015). DOI: 10.48550/arxiv.1505.05571 (cited on page 68).

[Ngu+15] L.-T. Nguyen, H. A. Schmidt, A. von Haeseler, and B. Q. Minh. "IQ-TREE: A Fast and Effective Stochastic Algorithm for Estimating Maximum-Likelihood Phylogenies". In: *Molecular Biology and Evolution* 32.1 (2015), pages 268–274. DOI: 10.1093/molbev/msu300 (cited on page 13).

[NHGRI25] National Human Genome Research Institute. *DNA Sequencing Costs*. 2025. URL: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data (cited on page 2).

[NHS25] National Library of Medicine (NHS). *GenBank and WGS Statistics*. 2025. URL: https://www.ncbi.nlm.nih.gov/genbank/statistics/ (cited on pages 1, 2).

[Nic+19] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello. "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale". In: *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press, 2019, pages 911–920. DOI: 10.1109/ipdps.2019.00099 (cited on pages 18, 19, 42, 44).

[NL79] M. Nei and W. H. Li. "Mathematical Model for Studying Genetic Variation in Terms of Restriction Endonucleases." In: *Proceedings of the National Academy of Sciences* 76.10 (1979), pages 5269–5273. ISSN: 1091-6490. DOI: 10.1073/pnas.76.10.5269 (cited on page 14).

[NWG98] Networking Working Group. *Internet Protocol, Version 6 Specification*. Request for Comments 2460. Cisco and Nokia, 1998 (cited on page 15).

[Nye08] T. M. W. Nye. "Trees of Trees: An Approach to Comparing Multiple Alternative Phylogenies". In: *Systematic Biology* 57.5 (2008). Edited by O. Gascuel, pages 785–794. ISSN: 1063-5157. DOI: 10.1080/10635150802424072 (cited on page 12).

[Obe+17] M. Obersteiner, A. Parra-Hinojosa, M. Heene, H. Bungartz, and D. Pflüger. "A Highly Scalable, Algorithm-Based Fault-Tolerant Solver for Gyrokinetic Plasma Simulations". In: *8th ACM Scalable Algorithms for Large-Scale Systems (ScalA)*. Edited by V. N. Alexandrov, A. Geist, and J. J. Dongarra. ACM Press, 2017, 2:1–2:8. DOI: 10.1145/3148226.3148229 (cited on pages 17, 19, 43).

[Oka+20] A. Okazaki, S. Yamazaki, I. Inoue, and J. Ott. "Population Genetics: Past, Present, and Future". In: *Human Genetics* 140.2 (2020), pages 231–240. ISSN: 1432-1203. DOI: 10.1007/s00439-020-02208-5 (cited on pages 1, 12, 24).

[Oli+24] A. Oliva, A. Kaphle, R. Reguant, L. M. F. Sng, N. A. Twine, Y. Malakar, et al. "Future-Proofing Genomic Data and Consent Management: A Comprehensive Review of Technology Innovations". In: *GigaScience* 13 (2024). ISSN: 2047-217x. DOI: 10.1093/gigascience/giae021 (cited on page 1).

[OS07] A. J. Oliner and J. Stearley. "What Supercomputers Say: A Study of Five System Logs". In: *37th IEEE/IFIP Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2007, pages 575–584. ISBN: 0-7695-2855-4. DOI: 10.1109/dsn.2007.103 (cited on page 17).

[Ou+92] C.-Y. Ou, C. A. Ciesielski, G. Myers, C. I. Bandea, C.-C. Luo, B. T. M. Korber, et al. "Molecular Epidemiology of HIV Transmission in a Dental Practice". In: *Science* 256.5060 (1992), pages 1165–1171. ISSN: 1095-9203. DOI: 10.1126/science.256.5060.1165 (cited on pages 1, 11).

[Pag+99] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical report 1999-66. Stanford InfoLab, 1999 (cited on page 43).

[Par+15] M. Parks, S. Subramanian, C. Baroni, M. C. Salvatore, G. Zhang, et al. "Ancient Population Genomics and the Study of Evolution". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 370.1660 (2015), page 20130381. ISSN: 1471-2970. DOI: 10.1098/rstb.2013.0381 (cited on pages 12, 24).

[Pat+12] N. Patterson, P. Moorjani, Y. Luo, S. Mallick, N. Rohland, Y. Zhan, et al. "Ancient Admixture in Human History". In: *Genetics* 192.3 (2012), pages 1065–1093. ISSN: 1943-2631. DOI: 10.1534/genetics.112.145037 (cited on page 14).

[Pen04] D. Penny. "Inferring Phylogenies". In: *Systematic Biology* 53.4 (2004), pages 669–670. ISSN: 1063-5157. DOI: 10.1080/10635150490468530 (cited on pages 1, 10, 12, 24).

[Pet16] B. M. Peter. "Admixture, Population Structure, and F-Statistics". In: *Genetics* 202.4 (2016), pages 1485–1501. ISSN: 1943-2631. DOI: 10.1534/genetics.115.183913 (cited on page 15).

[Pet21] B. M. Peter. "Modelling Complex Population Structure Using F-Statistics and Principal Component Analysis". In: *CoRR* (2021). DOI: 10.1101/2021.07.13.452141 (cited on page 15).

[Pie+09] W. H. Piel, L. Chan, M. J. Dominus, J. Ruan, R. A. Vos, and V. Tannen. "TreeBASE v. 2: A Database of Phylogenetic Knowledge". In: *e-BioSphere 2009* (2009) (cited on page 74).

[Pol16] A. Polukhin. *Boost.PFR*. 2016. URL: https://www.boost.org/doc/libs/1%5C%5F84%5C%5F0/doc/html/boost%5C%5Fpfr.html (cited on pages 90, 96).

[PSA11] F. Prinz, T. Schlange, and K. Asadullah. "Believe It or Not: How Much Can We Rely on Published Data on Potential Drug Targets?" In: *Nature Reviews Drug Discovery* 10.9 (2011), pages 712–712. ISSN: 1474-1784. DOI: 10.1038/nrd3439-c1 (cited on page 19).

[Rab00] R. Rabenseifner. “Automatic MPI Counter Profiling”. In: *42nd Cray User Group Conference*. Noorwik, The Netherlands, 2000 (cited on page 66).

[Rag+13] M. Raghavan, P. Skoglund, K. E. Graf, M. Metspalu, A. Albrechtsen, I. Moltke, et al. “Upper Palaeolithic Siberian Genome Reveals Dual Ancestry of Native Americans”. In: *Nature* 505.7481 (2013), pages 87–91. ISSN: 1476-4687. DOI: 10.1038/nature12736 (cited on page 15).

[Rei+09] D. Reich, K. Thangaraj, N. Patterson, A. L. Price, and L. Singh. “Reconstructing Indian Population History”. In: *Nature* 461.7263 (2009), pages 489–494. ISSN: 1476-4687. DOI: 10.1038/nature08365 (cited on page 15).

[Rei19] D. Reich. *Who We Are and How We Got Here. Ancient DNA and the New Science of the Human Past*. Oxford: Oxford University Press, 2019. 335 pages. ISBN: 9780198821250 (cited on pages 2, 12, 24).

[RF81] D. Robinson and L. Foulds. “Comparison of Phylogenetic Trees”. In: *Mathematical Biosciences* 53.1–2 (1981), pages 131–147. ISSN: 0025-5564. DOI: 10.1016/0025-5564(81)90043-2 (cited on pages 27, 74).

[RH09] U. Ramakrishnan and E. A. Hadly. “Using Phylochronology to Reveal Cryptic Population Histories: Review and Synthesis of 29 Ancient DNA Studies”. In: *Molecular Ecology* 18.7 (2009), pages 1310–1330. ISSN: 1365-294x. DOI: 10.1111/j.1365-294x.2009.04092.x (cited on pages 12, 24).

[Rie+10] M. Riester, C. Stephan-Otto Attolini, R. J. Downey, S. Singer, and F. Michor. “A Differentiation-Based Phylogeny of Cancer Subtypes”. In: *Public Library of Science (PLoS) Computational Biology* 6.5 (2010). Edited by W. S. Noble, e1000777. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1000777 (cited on page 11).

[Rin+16] S. Rinke, M. Butz-Ostendorf, M.-A. Hermanns, M. Naveau, and F. Wolf. “A Scalable Algorithm for Simulating the Structural Plasticity of the Brain”. In: *28th IEEE/SBC Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2016, pages 1–8. DOI: 10.1109/sbac-pad.2016.9 (cited on page 102).

[Roc06] S. Roch. “A Short Proof that Phylogenetic Tree Reconstruction by Maximum Likelihood Is Hard”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 3.1 (2006), pages 92–94. ISSN: 1545-5963. DOI: 10.1109/tcbb.2006.4 (cited on page 67).

[Roj+21] E. Rojas, E. Meneses, T. Jones, and D. Maxwell. “Understanding Failures Through the Lifetime of a Top-Level Supercomputer”. In: *Journal of Parallel and Distributed Computing* 154 (2021), pages 27–41. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2021.04.001 (cited on page 17).

[Rol+11] J. Rolland, M. W. Cadotte, J. Davies, V. Devictor, S. Lavergne, N. Mouquet, et al. “Using Phylogenies in Conservation: New Perspectives”. In: *Biology Letters* 8.5 (2011), pages 692–694. ISSN: 1744-957x. DOI: 10.1098/rsbl.2011.1024 (cited on pages 1, 11).

[Rom02] E. Roman. *A Survey of Checkpoint/Restart Implementations*. Technical report. Lawrence Berkeley National Laboratory, 2002 (cited on page 18).

[RRA11] R. W. Robey, J. M. Robey, and R. Aulwes. “In Search of Numerical Consistency in Parallel Programming”. In: *Parallel Computing* 37.4 (2011), pages 217–229. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.02.009 (cited on pages 3, 21, 64).

[RS18] N. Retzlaff and P. F. Stadler. “Phylogenetics Beyond Biology”. In: *Theory in Biosciences* 137.2 (2018), pages 133–143. ISSN: 1611-7530. DOI: 10.1007/s12064-018-0264-7 (cited on page 11).

[RS60] I. S. Reed and G. Solomon. “Polynomial Codes Over Certain Finite Fields”. In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pages 300–304. DOI: 10.1137/0108018 (cited on page 48).

[RTK20] P. Ralph, K. Thornton, and J. Kelleher. “Efficiently Summarizing Relationships in Large Samples: A General Duality Between Statistics of Genealogies and Genomes”. In: *Genetics* 215.3 (2020), pages 779–797. ISSN: 1943-2631. DOI: 10.1534/genetics.120.303253 (cited on pages 14, 25, 26, 29).

[Rue+21] M. Ruefenacht, D. Schafer, A. Skjellum, and P. V. Bangalore. “MPIs Language Bindings are Holding MPI Back”. In: *CoRR* (2021) (cited on pages 79, 84).

[Ryu+08] C.-K. Ryu, H.-J. Kim, S.-H. Ji, G. Woo, and H.-G. Cho. “Detecting and Tracing Plagiarized Documents by Reconstruction Plagiarism-Evolution Tree”. In: *8th IEEE International Conference on Computer and Information Technology (ICCIT)*. IEEE Press, 2008, pages 119–124. DOI: 10.1109/cit.2008.4594660 (cited on page 11).

[Sah+04] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. “Failure Data Analysis of a Large-Scale Heterogeneous Server Environment”. In: *34th IEEE/IFIP Dependable Systems and Networks ((DSN)*. IEEE Computer Society, 2004, page 772. ISBN: 0-7695-2052-9. DOI: 10.1109/dsn.2004.1311948 (cited on page 17).

[Sak09] S. Sakr. “XML Compression Techniques: A Survey and Comparison”. In: *Journal of Computer and System Sciences* 75.5 (2009), pages 303–322. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2009.01.004 (cited on page 26).

[San+19] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures. The Basic Toolbox*. 2019. ISBN: 9783030252090. DOI: 10.1007/978-3-030-25209-0 (cited on pages 15, 16, 50, 66, 70, 76, 77, 101, 105).

[San89] M. J. Sanderson. “Confidence Limits on Phylogenies: The Bootstrap Revisited”. In: *Cladistics* 5.2 (1989), pages 113–129. ISSN: 1096-0031. DOI: 10.1111/j.1096-0031.1989.tb00559.x (cited on page 76).

[Sat+23] H. Satam, K. Joshi, U. Mangrolia, S. Waghoo, G. Zaidi, S. Rawool, et al. “Next-Generation Sequencing Technology: Current Trends and Advancements”. In: *Biology* 12.7 (2023), page 997. ISSN: 2079-7737. DOI: 10.3390/biology12070997 (cited on page 1).

[Sau+19] G. Saunders, M. Baudis, R. Becker, S. Beltran, C. Béroud, E. Birney, et al. “Leveraging European Infrastructures to Access 1 Million Human Genomes by 2022”. In: *Nature Reviews Genetics* 20.11 (2019), pages 693–701. ISSN: 1471-0064. DOI: 10.1038/s41576-019-0156-9 (cited on pages 1, 2).

[SB25] C++ Reference. *Structured Binding Declaration*. 2025. URL: https://en.cppreference.com/w/cpp/language/structured%5C%5Fbinding (cited on page 89).

[Sch+18] K. Schwarze, J. Buchanan, J. C. Taylor, and S. Wordsworth. “Are Whole-Exome and Whole-Genome Sequencing Approaches Cost-Effective? A Systematic Review of the Literature”. In: *Genetics in Medicine* 20.10 (2018), pages 1122–1130. ISSN: 1098-3600. DOI: 10.1038/gim.2017.247 (cited on page 1).

[SDM10] J. Shalf, S. S. Dosanjh, and J. Morrison. "Exascale Computing Technology Challenges". In: *9th High Performance Computing for Computational Science (VECPAR)*. Edited by J. M. L. M. Palma, M. J. Daydé, O. Marques, and J. C. Lopes. Volume 6449. Lecture Notes in Computer Science. Springer, 2010, pages 1–25. DOI: 10.1007/978-3-642-19328-6_1 (cited on pages 3, 17, 42, 60, 106).

[SG10] B. Schroeder and G. A. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pages 337–351. DOI: 10.1109/tdsc.2009.4 (cited on page 17).

[SGB15] B. Steinbusch, A. Gaspar, and J. Brown. *rsmpi - MPI bindings for Rust*. 2015. URL: https://github.com/rsmpi/rsmpi (cited on page 91).

[Sha+19] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. "CRAFT: A Library for Easier Application-Level Checkpoint/Restart and Automatic Fault Tolerance". In: *IEEE Transactions on Parallel Distributed Systems* 30.3 (2019), pages 501–514. DOI: 10.1109/tpds.2018.2866794 (cited on pages 18, 19, 42, 44).

[She+20a] X.-X. Shen, Y. Li, C. T. Hittinger, X.-x. Chen, and A. Rokas. "An Investigation of Irreproducibility in Maximum Likelihood Phylogenetic Inference". In: *Nature Communications* 11.1 (2020). ISSN: 2041-1723. DOI: 10.1038/s41467-020-20005-6 (cited on pages 3, 20, 21, 64, 75, 81, 110).

[She+20b] Z. Shen, P. P. C. Lee, J. Shu, and W. Guo. "Cross-Rack-Aware Single Failure Recovery for Clustered File Systems". In: *IEEE Transactions on Dependable and Secure Computing* 17.2 (2020), pages 248–261. ISSN: 2160-9209. DOI: 10.1109/tdsc.2017.2774299 (cited on pages 64, 65, 67, 74, 75).

[Shi02] H. Shimodaira. "An Approximately Unbiased Test of Phylogenetic Tree Selection". In: *Systematic Biology* 51.3 (2002). Edited by N. Goldman, pages 492–508. ISSN: 1063-5157. DOI: 10.1080/10635150290069913 (cited on pages 68, 74).

[Shv+10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *26th IEEE Mass Storage Systems and Technologies (MSST)*. Ieee. 2010, pages 1–10 (cited on page 45).

[Sie24] D. Siebelt. "Faster Sorting of Aligned DNA-Read Files". Bachelor's Thesis. Karlsruhe Institute of Technology, 2024 (cited on page 125).

[SK06] P. Sung and H. Klein. "Mechanism of Homologous Recombination: Mediators and Helicases Take On Regulatory Functions". In: *Nature Reviews Molecular Cell Biology* 7.10 (2006), pages 739–750. ISSN: 1471-0080. DOI: 10.1038/nrm2008 (cited on pages 1, 12).

[SKC00] M. Schwab, N. Karrenbach, and J. Claerbout. "Making Scientific Computations Reproducible". In: *Computing in Science & Engineering* 2.6 (2000), pages 61–67. ISSN: 1521-9615. DOI: 10.1109/5992.881708 (cited on page 64).

[SKK20] A. Stamatakis, A. M. Kozlov, and A. M. Kozlov. "Efficient Maximum Likelihood Tree Building Methods". In: *Phylogenetics in the Genomic Era*. Edited by C. Scornavacca, F. Delsuc, and N. Galtier. 2020, 1.2:1–1.2:18 (cited on page 67).

[SL03] A. Stamatakis and T. Ludwig. "Phylogenetic Tree Inference on PC Architectures with AxML/PAxML". In: *17th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IPDPS-03. IEEE Computer Society, 2003, page 8. DOI: 10.1109/ipdps.2003.1213296 (cited on page 82).

[SMR24] B. Siklósi, G. R. Mudalige, and I. Z. Reguly. “Enabling Bitwise Reproducibility for the Unstructured Computational Motif”. In: *Applied Sciences* 14.2 (2024), page 639. ISSN: 2076-3417. DOI: 10.3390/app14020639 (cited on pages 3, 21, 66, 68).

[Sni+14] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, et al. “Addressing Failures in Exascale Computing”. In: *Journal of High Performance Computing Applications* 28.2 (2014), pages 129–173. DOI: 10.1177/1094342014522573 (cited on pages 3, 17, 42).

[Som+17] J. A. Somarelli, K. E. Ware, R. Kostadinov, J. M. Robinson, H. Amri, M. Abu-Asab, et al. “Phylo-Oncology: Understanding Cancer Through Phylogenetic Analysis”. In: *Biochimica et Biophysica Acta (BBA): Reviews on Cancer* 1867.2 (2017), pages 101–108. ISSN: 0304-419x. DOI: 10.1016/j.bbcan.2016.10.006 (cited on pages 1, 11).

[Sou+12] J. Soubrier, M. Steel, M. S. Lee, C. Der Sarkissian, S. Guindon, S. Y. Ho, et al. “The Influence of Rate Heterogeneity Among Sites on the Time Dependence of Molecular Rates”. In: *Molecular Biology and Evolution* 29.11 (2012), pages 3345–3358. ISSN: 0737-4038. DOI: 10.1093/molbev/mss140 (cited on page 7).

[SPW09] B. Schroeder, E. Pinheiro, and W.-D. Weber. “DRAM Errors in the Wild: A Large-Scale Field Study”. In: *ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS): Performance Evaluation Review* 37.1 (2009), pages 193–204. ISSN: 0163-5999. DOI: 10.1145/2492101.1555372 (cited on page 30).

[SR05] A. Sălcianu and M. Rinard. “Purity and Side Effect Analysis for Java Programs”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pages 199–215. ISBN: 9783540305798. DOI: 10.1007/978-3-540-30579-8_14 (cited on page 111).

[SRA25] National Library of Medicine (NHS). *SRA Database Growth*. 2025. URL: https://www.ncbi.nlm.nih.gov/sra/docs/sragrowth/ (visited on 01/16/2025) (cited on pages 1, 2).

[SRM16] G. M. Slota, S. Rajamanickam, and K. Madduri. “A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization”. In: *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2016, pages 293–302 (cited on page 102).

[SS+24] B. Stroustrup, H. Sutter, et al. *C++ Core Guidelines*. 2024. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html (cited on pages 84, 85, 88, 89, 91, 94, 99).

[SS16] R. Surendran and V. Sarkar. “Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis”. In: *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices* 51.10 (2016), pages 20–38. ISSN: 1558-1160. DOI: 10.1145/3022671.2984035 (cited on page 111).

[SS17] R. Schwartz and A. A. Schäffer. “The Evolution of Tumour Phylogenetics: Principles and Practice”. In: *Nature Reviews Genetics* 18.4 (2017), pages 213–229. ISSN: 1471-0064. DOI: 10.1038/nrg.2016.170 (cited on page 11).

[SS18] M. A. Supple and B. Shapiro. “Conservation of Biodiversity in the Genomics Era”. In: *Genome Biology* 19.1 (2018). ISSN: 1474-760x. DOI: 10.1186/s13059-018-1520-3 (cited on pages 1, 12, 24).

[SS23] P. Sanders and D. Seemaier. “Distributed Deep Multilevel Graph Partitioning”. In: *29th European Conference On Parallel and Distributed Computing (Euro-Par)*. Volume 14100. Lecture Notes in Computer Science. Springer, 2023, pages 443–457 (cited on pages 103, 106).

[SS24] D. Schreiber and P. Sanders. “MallobSat: Scalable SAT Solving by Clause Sharing”. In: *Journal of Artificial Intelligence Research* 80 (2024), pages 1437–1495. ISSN: 1076-9757. DOI: 10.1613/jair.1.15827 (cited on page 20).

[SST09] P. Sanders, J. Speck, and J. L. Träff. “Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan”. In: *Parallel Computing* 35.12 (2009), pages 581–594 (cited on page 99).

[Sta04] A. Stamatakis. “Distributed and Parallel Algorithms and Systems for Inference of Huge Phylogenetic Trees Based on the Maximum Likelihood Method”. PhD thesis. Technische Universität München, 2004 (cited on page 13).

[Sta10] A. Stamatakis. “Phylogenetic Search Algorithms for Maximum Likelihood”. In: *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications*. Edited by M. Elloumi and A. Y. Zomaya. Wiley, 2010, pages 547–577. ISBN: 9780470892107. DOI: 10.1002/9780470892107 (cited on page 7).

[Ste+15] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, et al. “Big Data: Astronomical or Genomical?” In: *Public Library of Science (PLoS) Biology* 13.7 (2015), e1002195. ISSN: 1545-7885. DOI: 10.1371/journal.pbio.1002195 (cited on page 1).

[Ste+20] P. Stefanelli, G. Faggioni, A. L. Presti, S. Fiore, A. Marchi, E. Benedetti, et al. “Whole Genome and Phylogenetic Analysis of Two SARS-CoV-2 Strains Isolated in Italy in January and February 2020: Additional Clues on Multiple Introductions and Further Circulation in Europe”. In: *Eurosurveillance* 25.13 (2020). DOI: 10.2807/1560-7917.es.2020.25.13.2000305 (cited on pages 1, 11).

[Ste22] C. Stelz. “Core-Count Independent Reproducible Reduce”. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2022 (cited on pages 79, 80, 125).

[Sto+16] H.-C. Stoeklé, M.-F. Mamzer-Bruneel, G. Vogt, and C. Hervé. “23andMe: A New Two-Sided Data-Banking Market Model”. In: *BMC Medical Ethics* 17.1 (2016). ISSN: 1472-6939. DOI: 10.1186/s12910-016-0101-9 (cited on page 2).

[Str00] C. Strachey. “Fundamental Concepts in Programming Languages”. In: *Higher-Order and Symbolic Computation* 13.1/2 (2000), pages 11–49. ISSN: 1388-3690. DOI: 10.1023/a:1010000313106 (cited on page 88).

[Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994 (cited on page 91).

[SU23] P. Sanders and T. N. Uhl. “Engineering a Distributed-Memory Triangle Counting Algorithm”. In: *37rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press, 2023, pages 702–712 (cited on page 106).

[Süß+20] T. Süß, L. Nagel, M.-A. Vef, A. Brinkmann, D. Feld, and T. Soddemann. “Pure Functions in C: A Small Keyword for Automatic Parallelization”. In: *Journal of Parallel Programming* 49.1 (2020), pages 1–24. ISSN: 1573-7640. DOI: 10.1007/s10766-020-00660-4 (cited on page 111).

[SV88] B. Schieber and U. Vishkin. “On Finding Lowest Common Ancestors: Simplification and Parallelization”. In: *SIAM Journal of Computing* 17.6 (1988), pages 1253–1262. DOI: 10.1137/0217079 (cited on page 29).

[Taj89] F. Tajima. “Statistical Method for Testing the Neutral Mutation Hypothesis by DNA Polymorphism.” In: *Genetics* 123.3 (1989), pages 585–595. ISSN: 1943-2631. DOI: 10.1093/genetics/123.3.585 (cited on page 14).

[Tal+21] D. Taliun, D. N. Harris, M. D. Kessler, J. Carlson, Z. A. Szpiech, et al. “Sequencing of 53831 Diverse Genomes from the NHLBI TOPMed Program”. In: *Nature* 590.7845 (2021), pages 290–299. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03205-y (cited on page 2).

[Tau+10] M. Taufer, O. Padron, P. Saponaro, and S. Patel. “Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on Gpus”. In: *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Volume 5462. IEEE Press, 2010, pages 1–9. DOI: 10.1109/ipdps.2010.5470481 (cited on pages 3, 21).

[TH14] K. Teranishi and M. A. Heroux. “Toward Local Failure Local Recovery Resilience Model using MPI-ULFM”. In: *21st European MPI Users’ Group Meeting (EuroMPI)*. Edited by J. J. Dongarra, Y. Ishikawa, and A. Hori. ACM Press, 2014, page 51. DOI: 10.1145/2642769.2642774 (cited on pages 18, 42).

[Tog+23] A. Togkousidis, O. M. Kozlov, J. Haag, D. Höhler, and A. Stamatakis. “Adaptive RAxML-NG: Accelerating Phylogenetic Inference under Maximum Likelihood using Dataset Difficulty”. In: *Molecular Biology and Evolution* 40.10 (2023). Edited by S. Bonatto. ISSN: 1537-1719. DOI: 10.1093/molbev/msad227 (cited on page 8).

[TRG05] R. Thakur, R. Rabenseifner, and W. Gropp. “Optimization of Collective Communication Operations in MPICH”. In: *Journal of High Performance Computing Applications* 19.1 (2005), pages 49–66. ISSN: 1741-2846. DOI: 10.1177/1094342005051521 (cited on pages 72, 73, 77, 78).

[TS18] A. Trefzer and A. Stamatakis. “Compressing Streams of Phylogenetic Trees”. In: *CoRR* (2018). DOI: 10.1101/440644 (cited on page 27).

[TW17] T. N. Theis and H.-S. P. Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19.2 (2017), pages 41–50. DOI: 10.1109/mcse.2017.29 (cited on page 2).

[Uhl+24a] T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, D. Seemaier, and P. Sanders. “KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI”. In: *High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pages 689–709. DOI: 10.1109/sc41406.2024.00050 (cited on pages 9, 83, 123).

[Uhl+24b] T. N. Uhl, M. Schimek, L. Hübner, D. Hespe, F. Kurpicz, C. Stelz, and P. Sanders. *KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI*. Technical report. Karlsruhe Institute of Technology, 2024, pages 1–21. DOI: 10.1109/sc41406.2024.00050 (cited on page 124).

[ULFM] *Document for a Standard Message-Passing Interface (Draft)*. MPI Forum, 2022 (cited on pages 19, 106).

[Vér+19] S. Véron, V. Saito, N. Padilla-García, F. Forest, and Y. Bertheau. “The Use of Phylogenetic Diversity in Conservation Biology and Community Ecology: A Common Base but Different Approaches”. In: *The Quarterly Review of Biology* 94.2 (2019), pages 123–148. ISSN: 1539-7718. DOI: 10.1086/703580 (cited on page 11).

[Vil+09] O. Villa, D. Chavarria-Miranda, V. Gurumoorthi, A. Marquéz, and S. Krishnamoorthy. "Effects of Floating-Point Non-Associativity on Numerical Computations on Massively Multithreaded Systems". In: *Cray User Group Meeting (CUG)*. Volume 3. 2009 (cited on pages 3, 21, 64, 68).

[VJG18] D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ Templates: The Complete Guide*. 2nd edition. Addison-Wesley, 2018 (cited on page 100).

[VM97] M. Vijay and R. Mittal. "Algorithm-Based Fault Tolerance: A Review". In: *Microprocessors and Microsystems* 21.3 (1997), pages 151–161. DOI: 10.1016/s0141-9331(97)00029-x (cited on page 17).

[Wei24] T. A. Weidmann. *Massively Parallel List-Ranking and Tree-Rooting*. en. Karlsruher Institut für Technologie (KIT), 2024. DOI: 10.5445/IR/1000173094 (cited on pages 107, 111).

[Wie+19] M. Wiesenberger, L. Einkemmer, M. Held, A. Gutierrez-Milla, X. Saez, and R. Iakymchuk. "Reproducibility, Accuracy and Performance of the Feltor Code and Library on Parallel Computer Architectures". In: *Computer Physics Communications* 238 (2019), pages 145–156. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.12.006 (cited on pages 3, 21, 38).

[WKW90] C. R. Woese, O. Kandler, and M. L. Wheelis. "Towards a Natural System of Organisms: Proposal for the Domains Archaea, Bacteria, and Eucarya". In: *Proceedings of the National Academy of Sciences* 87.12 (1990), pages 4576–4579. DOI: 10.1073/pnas.87.12.4576 (cited on pages 1, 11).

[WM03] T. Williams and B. Moret. "An Investigation of Phylogenetic Likelihood Methods". In: *3rd IEEE Bioinformatics and Bioengineering (BIBE)*. 2003, pages 79–86 (cited on page 13).

[WN91] G. C. Williams and R. M. Nesse. "The Dawn of Darwinian Medicine". In: *The Quarterly Review of Biology* 66.1 (1991), pages 1–22. ISSN: 1539-7718. DOI: 10.1086/417048 (cited on pages 1, 12, 24).

[Woh+22] A. W. Wohns, Y. Wong, B. Jeffery, A. Akbari, S. Mallick, R. Pinhasi, et al. "A Unified Genealogy of Modern and Ancient Genomes". In: *Science* 375.6583 (2022). ISSN: 1095-9203. DOI: 10.1126/science.abi8264 (cited on page 32).

[Won+23a] E. Wong, N. Bertin, M. Hebrard, R. Tirado-Magallanes, C. Bellis, W. K. Lim, et al. "The Singapore National Precision Medicine Strategy". In: *Nature Genetics* 55.2 (2023), pages 178–186. ISSN: 1546-1718. DOI: 10.1038/s41588-022-01274-x (cited on page 2).

[Won+23b] Y. Wong, A. Ignatieva, J. Koskela, G. Gorjanc, A. W. Wohns, et al. "A General and Efficient Representation of Ancestral Recombination Graphs". In: *arXiv* (2023). DOI: 10.1101/2023.11.03.565466 (cited on page 39).

[Wri50] S. Wright. "Genetical Structure of Populations". In: *Nature* 166.4215 (1950), pages 247–249. ISSN: 1476-4687. DOI: 10.1038/166247a0 (cited on page 14).

[Yan94] Z. Yang. "Maximum Likelihood Phylogenetic Estimation from DNA Sequences with Variable Rates over Sites: Approximate Methods". In: *Journal of Molecular Evolution* 39.3 (1994), pages 306–314. ISSN: 1432-1432. DOI: 10.1007/bf00160154 (cited on page 7).

[Yan95] Z. Yang. "A Space-Time Process Model for the Evolution of DNA Sequences". In: *Genetics* 139.2 (1995), pages 993–1005. ISSN: 1943-2631. DOI: 10.1093/genetics/139.2.993 (cited on page 7).

[Zah+12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th USENIX Networked Systems Design and Implementation (NSDI)*. Edited by S. D. Gribble and D. Katabi. USENIX Association, 2012, pages 15–28 (cited on page 45).

[Zei23] G. Zeitzmann. "Lightweight Checkpointing and Recovery for Iterative Converging Algorithms". Master's Thesis. Karlsruhe Institute of Technology, 2023 (cited on page 125).

[Zha+18] C. Zhang, M. Rabiee, E. Sayyari, and S. Mirarab. "ASTRAL-III: Polynomial Time Species Tree Reconstruction from Partially Resolved Gene Trees". In: *BMC Bioinformatics* 19.6 (2018), pages 15–30. DOI: 10.1186/s12859-018-2129-y (cited on page 27).

[ZJ12] N. Ziemert and P. R. Jensen. "Phylogenetic Approaches to Natural Product Structure Prediction". In: *Natural Product Biosynthesis by Microorganisms and Plants, Part C*. Elsevier, 2012, pages 161–182. DOI: 10.1016/b978-0-12-404634-4.00008-5 (cited on page 11).