Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.

Thèse n. XXXX 2014
présenté le 25 Juillet 2014
à la Faculté Informatique et Communications
laboratoire LAMP
programme doctoral en EDIC
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Aleksandar Prokopec

acceptée sur proposition du jury:

Prof Ola Svensson, président du jury
Prof Martin Odersky, directeur de thèse
Prof Douglas Lea, rapporteur
Prof Erik Meijer, rapporteur
Prof Viktor Kuncak, rapporteur

Lausanne, EPFL, 2014

Go confidently in the direction of your dreams.
Live the life you've imagined.
— Thoreau

To my parents and everything they gave me in this life.

Acknowledgements

Writing an acknowledgment section is a tricky task. I always feared omitting somebody really important here. Through the last few years, whenever I remembered a person that influenced my life in some way, I made a note to put that person here. I really hope I didn't forget anybody important. And by important I mean: anybody who somehow contributed to me obtaining a PhD in computer science. So get ready – this will be a long acknowledgement section. If somebody feels left out, he should know that it was probably by mistake. Anyway, here it goes.

First of all, I would like to thank my PhD thesis advisor Martin Odersky for letting me be a part of the Scala Team at EPFL during the last five years. Being a part of development of something as big as Scala was an amazing experience and I am nothing but thankful for it.

I want to thank my colleagues in the LAMP laboratory, who I had the chance to spend time with over the years – Antonio Cunei (good ol' Toni-boy), Ingo Maier (also known as Ingoman), Iulian Dragos, Lukas (Scalawalker) Rytz, Michel Schinz, Gilles Dubochet, Philipp Haller, Tiark Rompf, Donna Malayeri, Adriaan Moors, Hubert Plociniczak, Miguel Garcia, Vlad Ureche, Nada Amin, Denys Shabalin, Eugene Burmako, Sébastien Doeraene, Manohar Jonnalagedda, Sandro Stucki, Christopher Jan Vogt, Vojin Jovanovic, Heather Miller, Dmitry Petrashko and Samuel Grütter. I also want to thank my neighbouring office colleague Ružica Piskač for her constant positive attitude, regardless of my occasional mischiefs. I especially want to thank our secretary Danielle Chamberlain for her optimism and help in administrative matters, and our administrator Fabien Salvi for his technical support – the guy who had a solution for any computer-related problem I could think of.

A person that stood out during my time spent in LAMP is Phil Bagwell. Phil and I were on a similar wavelength – we were both passionate about data structures, whether mutable, persistent or concurrent. We spent many hours discussing ideas, challenging each other's approaches, finding and patching holes in our algorithms, and this ultimately lead to publishing great results. Although Phil is no longer with us at the time of writing this, I want to thank him for encouraging me to pursue my own ideas, and helping me get recognition.

Through the years, I was fortunate to supervise semester and master projects of several exceptional students. I feel that I have to mention at least a few of them. First, I want to thank Ngoc Duy Pham for the preliminary exploration of the techniques used in the ScalaMeter framework. It was a pleasure to collaborate with him, and together arrive at the conclusions that later served as foundations for ScalaMeter. Then, I want to thank Roman Zoller – his contributions to ScalaMeter no doubt attracted many new users. Also, I'd like to say thanks to Nicolas Stucki for his work on Scala multisets, Timo Babst for his contribution to the benchmark applications for ScalaBlitz, and Joel Rossier for his work on the MacroGL library.

When it comes to external collaborations, I'd like to thank the Akka team, prof. Kunle Olukotun and his team at Stanford, and most of all Nathan Bronson, for the discussions we've had on data structures and concurrent programming.

I especially want to thank Viktor Kunčak for his continuous help during these years. I have to thank Erik Meijer for an incredible five-day adventure we had in October 2013. Then, my thanks goes to Doug Lea for his support and, especially, the chat about seqlocks we had at ScalaDays 2011 – I think that was a turning point in my PhD. Conversations with Doug Lea almost always reveal a new surprising insight about computer science, and I enjoyed every single one we shared. I'm lucky to work in a similar field as him – in the swarms of compiler writers and type-systems people I was often surrounded with, talking to somebody who works on similar things felt like finding a long-lost friend. Finally, I would like to thank Viktor Kunčak, Erik Meijer and Doug Lea for agreeing to be on my thesis committee and for reviewing my thesis, as well as Ola Svensson for agreeing to be the thesis committee president.

Speaking of teachers, I cannot write an acknowledgement section without mentioning my piano teacher Jasna Reba. It might seem that what you taught me about music is not related to computer science in any way, but I would strongly disagree. Computer science, like music, has rules, patterns and loose ends with space to express yourself – research can be as artistic as music. If nothing else, those long hours of practice helped me build the willpower and persistence to work hard until reaching perfection. Thank you for that.

Although not directly related to computer science and this thesis, I'd like to thank Zlatko Varošanec for teaching the fundamentals of physics to a small group of gifted kids, myself included, in his physics cabinet. Your attention to details influenced me a lot during my life. Although I did not end up studying physics, I have to thank Dario Mičić for translating all those Russian physics exercise books, and Tihomir Engelsfeld for his support. When it comes to physics, I have to thank Juraj Szavits-Nossan, as well – without your selfless committment to teach us advanced physics every Saturday morning for four years without receiving any kind of compensation, me and many other people

would never have ended up on International Physics Olympiads.

I want to thank Senka Sedmak for teaching me how to think. Before we met in high school, mathematics was just a dry collection of methods used to compute something. That changed on a September day in the year 2000 when I saw your simple proof of Pythagora's theorem. At that point, things stopped being just about the *how*, and started being about the *why*. You showed me and generations of students how to reason in mathematics, and in life in general. Well, at least those who were willing to listen.

Marko Čupić is also a person who deserves to be mentioned here. Without your splendid Saturday morning initiative, many computer science students would get a diploma without the opportunity to learn Java, myself included. I want to thank Marin Golub for his counselling during my master thesis, and Željka Mihajlović for the great computer graphics course.

When it comes to my personal friends, I want to thank my good friend Gvero for all the good times we had since we simultaneously started working on our PhDs. After we arrived to Lausanne, my parents said how they hoped that this was the beginning of a long and great friendship. We soon started our apartment search. During this adventure I had to play the piano to retired Swiss ladies, we almost ended up taking the rent for an apartment filled with drug dealers, we got yelled at and threatened a couple of times, and, finally, we ended up having a bathtub in our bedroom for some reason. My parents' hopes came true – somewhere along the way we've become good friends. Let's keep it that way!

I want to thank my dear friend Žac who I know since childhood. You often engaged me in discussions about software technology – sometimes even too often! I want to thank other friends and acquaintances in my life as well: Boris and Mario for the discussions about the Zoo-kindergarten; Ida Barišić and her family for bearing me every morning at 9:30AM when I would barge in to play video games; Tin Rajković for showing me his GameBoy and Sega console; Krešimir for giving me his third release of the Hacker magazine (and protecting me from getting beaten a couple of times); Predrag Valožić for bringing those disquettes with Theme Park, and the words of wisdom he spoke after Theme Park would not run on my 486 PC; Julijan Vaniš for all those hours spent playing StarCraft; Damjan Šprem, Domagoj Novosel and Marko Mrkus for unforgettable days in high school; Dina Zjača for all the nice moments we had together, and encouraging me to pursue a PhD; Tomislav Car for believing in me; Josip and Eugen for teaching me about life; and Peđa Spasojević and Dražen Nadoveza for awesome lunchtimes, and weekends spent skiing. I want to thank Dalibor Mucko for all the good times we had together. I only regret that I never witnessed the advances you could have made in CS with your intellect, had you chosen to follow that path – I still hope someday you do. In any case, a separate, possibly longer thesis would be required to describe our crazy adventures. Enumerating all of them is quite possibly Turing-complete. Not to mention our brilliant e-mail discussions – they helped me get through every bad day of PhD. In fact, we're having one about Gvero just as I'm writing this.

I would like to thank my entire extended family, and particularly teta Miha (for excellent meals if nothing else). More than once did I found myself sitting at the table after lunch during these large family get-togethers, searching for a way to silently... disappear. Eventually, somebody would find me in some room in the attic writing some incomprehensible text on a piece of paper or frantically typing on my laptop. You guys were never being judgemental about me hacking some code when I should have had a normal human conversation and sit around the table like everybody else. Instead, you always showed me love and understanding. That kind of attitude – that's big. Now that I obtained a doctoral degree you might think that my social-contact-evading days are over for good. Sorry to dissapoint you, but you're out of luck. It will only get worse. You'll still be able to find me in the attic with an open laptop, though. Just look for places with a strong wi-fi connection.

Especially, I want to thank my grandmother. Her simple, yet spirited, character has often kept me positive, and inspired me throughout my life.

I want to thank Sasha for her endless patience and understanding, putting up with me as I'm writing this (it's 2AM in the morning), for standing by my side when I was down, and sharing my joy otherwise. I never met anybody like you, and I often ask myself what did I do to deserve the gift of your friendship.

And finally, I would like to thank my parents Marijana and Goran for everything they gave me and made possible for me, both from the materialistic and the non-materialistic point of view. I'm happy that I can say how you are my rolemodels in life. There is strong evidence that the system of values you installed in me works very well... in a majority of cases. One thing is certain, though – this system gave me guidance and a strong set of goals in life, made me who I am today. Mom, Dad, I cannot thank you enough for that.

Oh, and speaking of family members, I would like to thank Nera. So many times in these last 10 years you cheered me up when I needed it. I only wish I could spend more time with you to witness more of your amazing character. You're really one of a kind, you stupid mutt!

Lausanne, 12 Mai 2011 A. P.

Abstract

The data-parallel programming model fits nicely with the existing declarative-style bulk operations that augment collection libraries in many languages today. Data collection operations like reduction, filtering or mapping can be executed by a single processor or many processors at once. However, there are multiple challenges to overcome when parallelizing collection operations.

First, it is challenging to construct a collection in parallel by multiple processors. Traditionally, collections are backed by data structures with thread-safe variants of their update operations. Such data structures are called concurrent data structures. Their update operations require interprocessor synchronization and are generally slower than the corresponding single-threaded update operations. Synchronization costs can easily invalidate performance gains from parallelizing bulk operations such as mapping or filtering. This thesis presents a parallel collection framework with a range of data structures that reduce the need for interprocessor synchronization, effectively boosting data-parallel operation performance. The parallel collection framework is implemented in Scala, but the techniques in this thesis can be applied to other managed runtimes.

Second, most concurrent data structures can only be traversed in the absence of concurrent modifications. We say that such concurrent data structures are *quiescently consistent*. The task of ensuring quiescence falls on the programmer. This thesis presents a novel, lock-free, scalable concurrent data structure called a *Ctrie*, which supports a linearizable, lock-free, constant-time snapshot operation. The Ctrie snapshot operation is used to parallelize Ctrie operations without the need for quiescence. We show how the linearizable, lock-free, constant-time snapshot operation can be applied to different concurrent, lock-free tree-like data structures.

Finally, efficiently assigning parts of the computation to different processors, or *scheduling*, is not trivial. Although most computer systems have several identical CPUs, memory hiearchies, cache-coherence protocols and interference with concurrent processes influence the effective speed of a CPU. Moreover, some data-parallel operations inherently require more work for some elements of the collection than others – we say that no data-parallel operation has a *uniform workload* in practice. This thesis presents a novel technique for parallelizing highly irregular computation workloads, called the *work-stealing tree scheduling*. We show that the work-stealing tree scheduler outperforms other schedulers when parallelizing highly irregular workloads, and retains optimal performance when parallelizing more uniform workloads.

Concurrent algorithms and data structure operations in this thesis are linearizable and lock-free. We present pseudocode with detailed correctness proofs for concurrent data structures and algorithms in this thesis, validating their correctness, identifying linearization points and showing their lock-freedom.

Key words: parallel programming, data structures, data-parallelism, parallelization, concatenation, scheduling, atomic snapshots, concurrent data structures, persistent data structures, work-stealing, linearizability, lock-freedom

Zusammenfassung

Daten-parallele Programmierung integriert sich gut in die existierenden deklarativen Bulk-Operationen der Collection-Bibliotheken vieler Sprachen. Collection-Operationen wie Reduktion, Filterung, oder Transformation können von einem einzigen oder von mehreren Prozessoren gleichzeitig ausgeführt werden. Bei der Parallelisierung von Collection-Operationen gibt es jedoch einige Herausforderungen zu meistern.

Erstens ist es schwierig Collections parallel mithilfe mehrerer Prozessoren zu erzeugen. Traditionellerweise sind Collections implementiert mithilfe von Datenstrukturen, zu deren Update-Operationen es Varianten gibt, die threadsafe sind. Solche Datenstrukturen werden nebenläufige Datenstrukturen genannt. Deren Update-Operationen benötigen Inter-Prozessor-Synchronisierung und sind im Generellen weniger effizient als die korrespondierenden sequentiellen Update-Operationen. Die Synchronisierungskosten können etwaige Leistungsgewinne der Parallelisierung von Bulk-Operationen, wie Transformieren oder Filtern, einfach zunichte machen. Diese Dissertation zeigt ein Framework für parallele Collections mit einer Auswahl an Datenstrukturen, die benötigte Inter-Prozessor-Synchronisierung minimieren, und so die Leistung daten-paralleler Operationen effektiv steigern. Das Framework für parallele Collections ist in Scala implementiert, die Techniken in dieser Dissertation können jedoch auf andere Laufzeitumgebungen angewandt werden.

Zweitens kann über die meisten nebenläufigen Datenstrukturen nur in Abwesenheit nebenläufiger Änderungen iteriert werden. Wir nennen solche nebenläufigen Datenstrukturen *leerlauf-konsistent*. Die Aufgabe für Leerlauf zu Sorgen fällt auf den Programmierer. Diese Dissertation zeigt eine neuartige, lock-free Snapshot-Operation mit konstanter Zeitkomplexität. Die Ctrie Snapshot-Operation wird zur Parallelisierung von Ctrie-Operationen verwendet, ohne Leerlauf zu benötigen. Wir zeigen, wie die linearisierbare, lock-free Snapshop-Operation mit konstanter Zeitkomplexität auf verschiedene nebenläufige, lock-free baumartige Datenstrukturen angewandt werden kann.

Schliesslich ist die Zuweisung von Teilen der Berechnung zu verschiedenen Prozessoren, oder *Scheduling*, nicht trivial. Obwohl die meisten Computersysteme mehrere identische CPUs haben, beeinflussen Speicherhierarchien, Cache-Koherenz-Protokolle, und Interferenz nebenläufiger Prozesse die effektive Leistung einer CPU. Ausserdem benötigen einige daten-parallele Operationen inherent mehr Aufwand für bestimmte Elemente einer Collection – wir sagen, dass keine daten-parallele Operation in der Praxis eine *gleichverteilte Arbeitslast* hat. Diese Dissertation zeigt eine neuartige Methode zur Parallelisierung hochgradig ungleichverteilter Arbeitslasten, genannt *Work-Stealing Tree Scheduling.* Wir

zeigen, dass der Work-Stealing Tree Scheduler andere Scheduler bei der Parallelisierung hochgradig ungleichverteilter Arbeitslasten leistungsmässig übertrifft, und seine optimale Leistung bei der Parallelisierung gleichverteilterer Arbeitslasten beibehält.
Die nebenläufigen Algorithmen und Datenstrukturen-Operationen in dieser Dissertation sind linearisierbar und lock-free. Wir zeigen Pseudocode mit detailierten Korrektheits-Beweisen für die nebenläufigen Datenstrukturen und Algorithmen in dieser Dissertation, um deren Korrektheit zu bestätigen, Linearisierungspunkte zu identifizieren, und deren Eigenschaft lock-free zu sein zu zeigen.

Stichwörter: Daten-parallele Programmierung, Datenstrukturen, Parallelisierung, Scheduling, Snapshots, nebenläufige Datenstrukturen, Work-Stealing, Linearisierbarkeit, Lock-Freiheit

Contents

List of Figures

1 Introduction

It is difficult to come up with an original introduction for a thesis on parallel computing in an era when the same story has been told over and over so many times. So many paper abstracts, technical reports, doctoral thesis and funding requests begin in the same way – the constant improvements in processor technology have reached a point where the processor clock speed, or the running frequency, can no longer be improved. This once driving factor of the computational throughput of a processor is now kept at a steady rate of around 3.5 GHz. Instead of increasing the processor clock speed, major commercial processor vendors like Intel and AMD have shifted their focus towards providing multiple computational units as part of the same processor, and named the new family of central processing units *multicore processors*. These computer systems, much like their older multiprocessor cousins, rely on the concept of shared-memory in which every processor has the same read and write access to the part of the computer called *the main memory*.

Despite this clichéd story that every parallel computing researcher, and since recently the entire developer community, heard so many times, the shift in the processor technology has resulted in a plethora of novel and original research. Recent architectural developments spawned incredibly fruitful areas of research and helped start entire new fields of computer science, as well as revived some older research from the end of the 20th century that had previously quieted down. The main underlying reason for this is that, while developing a program that runs correctly and efficiently on a single processor computer is challenging, creating a program that runs correctly on many processors is magnitudes of times harder with the programming technology that is currently available. The source of this difficulty lies mostly in the complexity of possible interactions between different computations executed by different processors, or processor cores. These interactions manifest themselves in the need for different processors to communicate, and this communication is done using the above-mentioned main memory shared between different processors.

There are two main difficulties in programming a multiprocessor system in which processors communicate using shared-memory. The first is achieving the correctness of an algorithm or a program. Parallel programs that communicate using shared-memory usually produce outputs that are non-deterministic. They may also contain subtle, hard-to-reproduce errors due to this non-determinism, which occasionally cause unexpected program outputs or even completely corrupt the program state. Controlling non-determinism and concurrency errors does not rely only on the programmer's correct understanding of the computational problem, the conceptual solution for it and the implementation of that solution, but also on the specifics of the underlying hardware and memory model.

The second difficulty is in achieving consistent performance across different computer architectures. The specific features of the underlying hardware, operating systems and compilers may cause a program with optimal performance on one machine to run inefficiently on another. One could remark that both these difficulties are present in classical single-threaded programming. Still, they seem to affect us on a much larger scale in parallel programming.

To cope with these difficulties, a wide range of programming models, languages and techniques have been proposed through the years. While some of these models rise briefly only to be replaced with new ideas, several seem to have caught on for now and are becoming more widely used by software developers. There is no single best among these programming models, as each approach seems to fit better for a different class of programs. In this thesis we focus on the data-parallel programming model.

Modern software relies on high-level data structures. A data structure is a set of rules on how to organize data units in memory in a way such that specific operations on that data can be executed more efficiently. Different types of data structures support different operations. *Data collections* (or *data containers*) are software modules that implement various data structures. Collections are some of the most basic building blocks of any program, and any serious general purpose language comes with a good collections library. Languages like Scala, Haskell, C#, and Java support *bulk operations* on collections, which execute a certain computation on every element of the collection independently, and compute a final result from all the computations. Bulk operations on collections are highly parametric and can be adapted and composed into different programs.

This high degree of genericity does not come without a cost. In many high-level languages bulk operations on collections can be quite far from optimal, hand-written code – we explore the reasons for this throughout this thesis. In the past this suboptimality was disregarded with the hope that a newer processor with a higher clock will solve all the performance problems. Today, the case for optimizing collection bulk operations feels more important.

Another venue of achieving more efficient collection operations is the data-parallel computing model, which exploits the presence of these bulk operations. Bulk collection operations are an ideal fit for the data-parallel programming model since both execute operations on elements in parallel.

In this thesis we describe how to implement a generic data-parallel computing framework running on a managed runtime inside a host language – we cover a wide range of single-threaded, concurrent thread-safe and reactive data structures to show how data-parallel operations can be executed on their data elements. In doing so, we introduce the necessary abstractions required for the generic data-parallel framework design, the implementations of those abstractions in the form of proper algorithms and data structures, and compile- and runtime techniques required to achieve optimal performance. The specific managed runtime we focus on is the JVM, and the host language is Scala. We note that the algorithms and data structures described in this thesis are applicable to other managed runtimes. For programming languages that compile directly into native code most of the data structures can be reused directly with minor modifications.

In this thesis we address the following problems:

- What are the minimal abstractions necessary to generically implement a wide range of data-parallel operations and collections, and how to implement these abstractions efficiently?
- What are good data structures for supporting data-parallel computing?
- How can the data-parallel programming model be applied to data structures that allow concurrent modifications?
- How can the data-parallel programming runtime system efficiently assign work to different processors?

1.1 The Data Parallel Programming Model

The driving assumption behind the data-parallel programming model is that there are many individual data elements that comprise a certain data set, and a computation needs to be executed for each of those elements independently. In this programming model the parallel computation is declared by specifying the data set and the parallel operation. In the following example the data set is the range of numbers from 0 until 1000 and the parallel operation is a `foreach` that increments some array entry by 1:

```
(0 until 1000).par.foreach(i => array(i) += 1)
```

A *nested data-parallel programming model* allows nesting data-parallel operations arbitrarily – another `foreach` invocation can occur in the closure passed to the `foreach` invocation above. The data-parallel programming model implementation in this thesis allows nested data parallelism.

Generic data-parallelism refers to the ability of the data-parallel framework to be used for different data structures, data element types and user-specified operators. In the example above we chose the Scala `Range` collection to represent our array indices, but it could really have been any other collection. In fact, instead of indices this collection could have contained `String` keys in which case the lambda passed to the `foreach` method (i.e. the operator) could have accessed a map of strings and integers to increase a specific value. The same `foreach` operation should apply to any of these collections and data-types. Many modern data-parallel frameworks, like Intel TBB, STAPL, Java 8 Streams or PLinq, are generic on several levels. However, there are many data-parallel frameworks that are not very generic. For example, the basic MPI implementation has a limited predefined set of parallel operators and data types for its reduce operation, and some GPU-targeted data-parallel frameworks like C++ AMP are very limited at handling arbitrary data structures. The framework in this thesis is fully generic in terms of data structures it supports, data element types in those data structures and user-defined operators for different parallel operations.

1.2 Desired Algorithm Properties

When it comes to algorithm properties, a property usually implicitly agreed upon is their *correctness* – given a set of inputs, the algorithms should produce outputs according to some specification. For concurrent algorithms, the outputs also depend on the possible interactions between concurrent processes, in addition to the inputs. In this thesis we always describe what it means for an algorithm or an abstraction to be correct and strive to state the specification as precisely as possible.

A standard methodology to assess the quality of the algorithms is by showing their *algorithmic complexity*. The running time, memory or energy requirements for a given size of the problem are expressed in terms of the big O notation. We will state the running times of most algorithms in terms of the big O notation.

Concurrent algorithms are special in that they involve multiple parallel computations. Their efficiency is dictated not by how well a particular parallel computation works, but how efficient they are in unison, in terms of memory consumption, running time or something else. This is known as *horizontal scalability*. There is no established theoretical model for horizontal scalability. This is mainly due to the fact that it depends on a large number of factors, many of which are related to the properties of the underlying memory model, processor type, and, generally, the computer architecture characteristics such as the processor-memory throughput or the cache hierarchy. We mostly rely on benchmarks to evaluate horizontal scalability.

There are two additional important properties that we will require from the concurrent algorithms in this thesis. The first is linearizability and the second is lock-freedom.

1.2.1 Linearizability

Linearizability is an important property of operations that may be executed concurrently [Herlihy and Wing(1990)]. An operation executed by some processor is *linearizable* if the rest of the system observes the corresponding system state change as if it occured instantaneously at a single point in time after the operation started and before it finished. This property ensures that all the linearizable operations in the program have a mutual order observed in the same way by the entire system.

While concurrent operations may generally have weaker properties, linearizability is particularly useful to have, since it makes reasoning about the programs easier. Linearizability can be proven easily if we can identify a single instruction or sub-operation which changes the data structure state and is known to be itself linearizable. In our case, we will identify CAS instructions as linearization points.

1.2.2 Non-Blocking Algorithms and Lock-Freedom

In the context of this thesis, lock-freedom [Herlihy and Shavit(2008)] is another important property. An operation *op* executed by some processor P is lock-free if and only if during an invocation of that operation *op* some (potentially different) processor Q completes some (potentially different) invocation of *op* after a finite number of steps. Taking linearizability into account, completing an operation means reaching its linearization point. This property guarantees system-wide progress, as at least one thread must always complete its operations. Lock-free algorithms are immune to failures and preemptions. Failures inside the same runtime do not occur frequently, and when they happen they usually crash the entire runtime, rather than a specific thread. On the other hand, preemption of threads holding locks can easily compromise performance.

More powerful properties of concurrent algorithms exist, such as *wait-freedom* that guarantees that every operation executed by every processor completes in a finite number of execution steps. We are not interested in wait-freedom or other termination properties in the context of this thesis – lock-freedom seems to be an adequate guarantee in practice, and wait-freedom comes with a much higher price when used with primitives like CAS, requiring $O(P)$ space for P concurrently executing computations [Fich et al.(2004)Fich, Hendler, and Shavit].

1.3 Implications of Using a Managed Runtime

We rely on the term *managed code* to refer to code that requires a special supporting software to run as a program. In the same note, we refer to this supporting software as a *managed runtime*. This thesis studies efficient data structures and algorithms for data-parallel computing in a managed runtime, so understanding the properties of a managed runtime is paramount to designing good algorithms. The assumptions we make in this thesis are subject to these properties.

We will see that a managed runtime offers infrastructure that is an underlying building block for many algorithms in this thesis. However, while a managed runtime is a blessing for concurrent algorithms in many ways, it imposes some restrictions and limits the techniques for designing algorithms.

1.3.1 Synchronization Primitives

Most modern runtimes are designed to be *cross-platform*. The same program code should run in the same way on different computer architectures, processor types, operating systems and even versions of the managed runtime – we say that a combination of these is a specific *platform*. The consequences of this are twofold. First, runtimes aim to provide a standardized set of programming primitives and work in exactly the same way on any underlying platform. Second, because of this standardization the set of programming primitives and their capabilities are at least as limited as on any of these platforms. The former makes a programmer's life easier as the application can be developed only on one platform, while the second makes the programming task more restrictive and limited.

In the context of concurrent lock-free algorithms, primitives that allow different processors to communicate to each other and agree on specific values in the program are called *synchronization primitives*. In this section we will overview the standardized set of synchronization primitives for the JVM platform – we will rely on them throughout the thesis. A detailed overview of all the concurrency primitives provided by the JDK is presented by Goetz et al. [Goetz et al.(2006)Goetz, Bloch, Bowbeer, Lea, Holmes, and Peierls].

On the JVM different parallel computations are not assigned directly to different processors, but multiplexed through a construct called a *thread*. Threads are represented as special kinds of objects that can be created programatically and started with the `start` method. Once a thread is started, it will eventually be executed by some processor. The JVM delegates this multiplexing process to the underlying operating system. The thread executes its `run` method when it starts. At any point, the operating system may temporarily cease the execution of that thread and continue the execution later, possibly on another processor. We call this *preemption*. A thread may wait for the completion of another thread by calling its `join` method. A thread may call the `sleep` method to postpone execution for approximately the specified time. An important limitation is that it is not possible to set the affinity of some thread – the JVM does not allow choosing the processor for the thread, similar how it does not allow specifying which region of memory is allocated when a certain processor allocates memory. Removing these limitations would allow designing more efficient algorithms on, for example, NUMA systems.

Different threads do not see memory writes by other threads immediately after they occur. This limitation allows the JVM runtime to optimize programs and execute them more efficiently. All memory writes executed by a thread that has stopped are visible to the threads that waited for its completion by invoking its `join` method.

To allow different threads to communicate and exchange data, the JVM defines the `synchronized` block to protect critical sections of code. A thread that calls `synchronized` on some object o has a guarantee that it is the only thread executing the critical section for that object o. In other words, invocations of `synchronized` on the same object are serialized. Other threads calling `synchronized` on o have to wait until there is no other thread executing `synchornized` on o. Additionally, all the writes by other threads in the previous corresponding `synchronized` block are visible to any thread entering the block.

The `synchronized` blocks also allows threads to notify each other that a certain condition has been fulfilled. This is done with the `wait` and `notify` pair of methods. When a thread calls `wait` on an object, it goes to a dormant state until another thread calls `notify`. This allows threads to check for conditions without spending processor time to continuously poll if some condition is fulfilled – for example, it allows implementing thread pools.

The JVM defines *volatile variables* the writes to which are immediately visible to all other threads. In Java these variables are defined with the `volatile` keyword before the declaration, and in Scala with the `@volatile` annotation. When we present pseudocode in this thesis, we will denote all volatile variables with the keyword `atomic`. This is not a standard Scala keyword, but we find that it makes the code easier to understand. Whenever we read a volatile variable `x`, we will use the notation `READ(x)` in the pseudocode, to make it explicit that this read is atomic, i.e. it can represent a linearization point. When writing the value `v` to the volatile variable, we will use the notation `WRITE(x, v)`.

A special *compare-and-set* or CAS instruction atomically checks if the target memory location is equal to the specified expected value and, if it is, writes the new value, returning `true`. Otherwise, it returns `false`. It is basically equivalent to the corresponding synchronized block, but more efficient on most computer architectures.

A conceptually equally expressive, but in practice more powerful pair of instructions load-linked/store-conditional LL/SC is not available in mainstream computer architectures.

Even though this instruction pair would simplify many algorithms in this thesis, we cannot use it on most managed runtimes. LL/SC can be simulated with CAS instructions, but this is prohibitively expensive.

We also assume that the managed runtime does not allow access to hardware counters or times with submicrosecond precision. Access to these facilities can help estimate how much computation steps or time a certain computation takes. Similarly, the JVM and most other managed runtimes do not allow defining custom interrupts that can suspend the main execution of a thread and have the thread run custom interrupt code. These fundamental limitations restrict us from improving the work-stealing scheduler in Chapter 5 further.

1.3.2 Managed Memory and Garbage Collection

A defining feature of most managed runtimes is *automatic memory management*. Automatic memory management allows programmers to dynamically allocate regions of memory for their programs from a special part of memory called a *heap*. Most languages today agree on the convention to use the `new` keyword when dynamically allocating objects. What makes automatic memory management special is that the `new` invocation that produced an object does not need to be paired with a corresponding invocation that frees this memory region. Regions of allocated memory that are no longer used are freed automatically – to do this, heap object reachability analysis is done at program runtime and unreachable allocated memory regions are returned to the memory allocator. This mechanism is called *garbage collection* [Jones and Lins(1996)]. While garbage collection results in a non-negligible performance impact, particularly for concurrent and parallel programs, it simplifies many algorithms and applications.

Algorithms in this thesis rely on the presence of accurate automatic memory management. In all code and pseudocode in this thesis objects and data structure nodes that are dynamically allocated on the heap with the `new` keyword are never explicitly freed. In some cases this code can be mended for platforms without automatic memory management by adding a corresponding `delete` statement. In particular, this is true for most of the code in Chapters 2 and 3.

There are some less obvious and more fundamental ways that algorithms and data structures in this thesis depend on a managed runtime. Lock-free algorithms and data structures in this thesis rely on the above-mentioned CAS instruction. This synchronization primitive can atomically change some location in memory. If this location represents a pointer p to some other object a (i.e. allocated region) in memory, then the CAS instruction can succeed in two scenarios. In the first scenario, the allocated region a at p is never deallocated, thread T1 attempts a CAS on p with the expected value a and succeeds. In the second scenario, some other thread T2 changes p to some other value b, deallocates object at address a, then allocates the address a again, and writes a to p – at this point the original thread T1 attempts the same CAS and succeeds. There is fundamentally no way for the thread T1 to tell these two scenarios apart. This is known as the *ABA* problem [Herlihy and Shavit(2008)] in concurrent lock-free computing, and it impacts the correctness of a lock-free algorithm.

The benefit of accurate automatic garbage collection is the guarantee that the object at address a above cannot be deallocated as long as some thread is about to call CAS with a as the expected value – if it were deallocated, then no thread would have a pointer with the address a, hence no stale CAS could be attempted with address a as the expected value. As a result, the second scenario described above can never happen – accurate automatic memory management helps ensure the *monotonicity* of CAS writes to memory locations that contain address pointers.

Modifying lock-free algorithms for unmanaged runtimes without accurate garbage collection remains a challenging task. In some cases, such as the lock-free work-stealing tree scheduler in Chapter 5, there is a point in the algorithm when it is known that there some or all nodes may be deallocated. In other cases, such as lock-free data structures in Chapter 4 there is no such guarantee. Michael studied approaches like hazard pointers [Michael(2004)] to solve these issues, and pointer tagging is helpful in specific cases, as shown by Harris [Harris(2001)]. Still, this remains an open field of study.

1.3.3 Pointer Value Restrictions

Most managed runtimes do not allow arbitrary pointer arithmetic or treating memory addresses as representable values – memory addresses are treated as abstract data types created by the keyword `new` and can be used to retrieve fields of object at those addresses according to their type. There are several reasons for this. First, if programmers are allowed to form arbitrary addresses, they can corrupt memory in arbitrary ways and compromise the security of programs. Second, modern garbage collectors do not just free memory regions, but are allowed to move objects around in the interest of better performance – the runtime value of the same pointer in a program may change without the program knowing about it. Storing the addresses in ways not detectable to a garbage collector (e.g. in a file) means that the programmers could reference a memory region that has been moved somewhere else.

The algorithms in this thesis will thus be disallowed from doing any kind of pointer arithmetic, such as pointer tagging. Pointer tagging is particularly useful when atomic changes depend on the states of several parts of the data structure, but we will restrict ourselves to a more narrow range of concurrent algorithm techniques.

There are exceptions like the D programming language, or the `Unsafe` package in Java and the `unsafe` keyword in C#, that allow using arbitrary pointer values and some form of pointer arithmetic, but their use is limited. We assume that pointer values can only be formed by the `new` keyword and can only be used to dereference objects. We call such pointers *references*, but use the two terms interchangeably.

1.3.4 Runtime Erasure

Managed runtimes like the JVM do directly not allow type polymorphism, which is typically useful when developing frameworks that deal with data. For example, Java and Scala use *type erasure* to compile a polymorphic class such as `class ArrayBuffer[T]`. After type-checking, the type parameter `T` is simply removed and all its occurrences are replaced with the `Object` type. While this is an extremely elegant solution, its downside is that on many runtimes it implies creating objects when dealing with primitives. This is the case with Scala and the JVM, so we rely on compile-time techniques like Scala type-specialization [Dragos and Odersky(2009)] and Scala Macros [Burmako and Odersky(2012)] to generate more efficient collections versions for primitive types like integers.

1.3.5 JIT Compilation

Scala code is compiled into Java bytecode, which in turn is compiled into platform-specific machine code by the *just-in-time compiler*. This JIT compilation occurs after the program already runs. To avoid compiling the entire program and slowing execution too much, this is done adaptively in steps. As a result, it takes a certain amount of *warmup time* until the program is properly compiled and optimized. When measuring running time on the JVM and evaluating algorithm efficiency, we need to account for this warmup time [Georges et al.(2007)Georges, Buytaert, and Eeckhout].

1.4 Terminology

Before we begin studying data-parallelism, we need to agree on terminology. In this section we present of overview of terms and names used throughout this thesis.

We mentioned multicore processors at the very beginning and hinted that they are in principle different from multiprocessors. From a perspective of a data-parallel framework in this thesis, these two abstractions really mean the same thing – a part of the computer system that can perform arbitrary computation on its own. This is why we call any separate computational unit *a processor*.

We have already defined a runtime as computer software that allows a specific class of programs to run. The homonym *runtime* refers to the time during which a program executes along with its entire state. We will usually refer to the latter definition of the term runtime.

We have already defined variables that contain memory addresses as pointers, and their restricted form as references. We will use these terms interchangeably, as the distinction is usually not very important. Note that references are not exactly the C++ language references, but restricted pointers in a broader sense.

We will often use the term *abstraction* to refer to some isolated program state with clearly defined operations that access and possibly change that state. For example, a *stack* is an abstraction – it defines a string of elements along with a pair of operations *push* and *pop* that extend this string by a single element and remove the last element in the string, respectively. We will call a concrete program that allocates memory for this state and manipulates it with a set of program subroutines an *abstraction implementation*. A stack may be implemented with an array or a linked list – an abstraction can have many implementations. A *data structure* is a specific class of allowed arrangements of memory regions and contents of those regions, along with the operations to access them. Throughout the thesis we often use these three terms interchangeably, but we make sure to use the correct term to disambiguate when necessary.

A basic abstraction in this thesis is a *collection*. A collection usually implies the element traversal operation. In Scala a collection is represented by the `Traversable` type, while in Java it is the `Iterable` type. There are many more specific collections. A *pool* implies the existence of an *add* operation – upon invoking it with some element x, a subsequent traversal should reflect that x was added. A *set* also allows querying if a specific element is a part of it, and removing the element if so. A *map* allows adding pairs of keys and values, and later retrieving values associated with specific keys. A *sequence* is a collection that defines a specific integral index for each element such that indices form a contiguous range from 0 to $n - 1$, where n is the number of elements. We refer to the operation that retrieves an element associated with an index as *indexing*. A *priority queue* is a pool that allows retrieving the previously added element that is the smallest according to some total ordering. There are other collection abstractions, but we will mostly be focusing on these.

Mutable data structures support operations that can change their state after their construction. These operations are called *destructive operations* or *modification operations*. A special kind of mutable data structures are called *concurrent* – their modification operations can be safely invoked by several processors concurrently without the risk of corrupting the state of the data structure.

Immutable data structures are never modified in memory after they are constructed – after the data structure is initially layed out in memory, neither this layout, nor its contents are changed. *Functional data structures* are data structures whose operations always return the same values for the same inputs [Okasaki(1998)] – note that their memory contents can be modified, but programs cannot observe this. We call both data structure classes *persistent data structures*. Instead of modifying it, we use an immutable data structure to produce a new different version with the modification.

Immutable data structures can be used both *ephemerally* and *persistently*. In the ephemeral use, whenever a persistent data structure is used to produce a new version, the old version is discarded and never used by the program again. In the persistent use all the old versions are kept, and can be repetitively used in operations. Ephemeral and persistent use of the same persistent data structure may result in very different asymptotic running times – usually the distinction manifests itself as a difference in amortized running time [Okasaki(1998)]. Mutable data structures are always used ephemerally.

Part of this thesis focuses on *combining* data structures – efficiently converting many data structures produced by different processors into a single data structure containing the union of their elements. Depending on the data structure in question, we will call this process differently. *Concatentation* is an operation that given two input sequences produces a third sequence composed of the elements in the first input sequence and the elements of the second input sequence with their indices shifted. In essence, concatenation works on sequences and appends one sequence at the end of another. *Merge* is an operation that given two pools, sets, maps or priority queues returns a new collection composed of the elements of both input collections. Merging is more closely related to the set union operation than concatenation, as it does not assume that elements are ordered according to some indices as they are in a sequence.

1.5 Intended Audience

The hope is that this thesis will be useful to practitioners seeking to gain knowledge about designing a data-parallel programming module and augmenting their language with data-parallel collection operations. This thesis is, first and foremost, intended as a practical manual on how to implement the proper abstractions, corresponding data structures and scheduling algorithms for efficient data-parallel computations on multiprocessor architectures. We strived to produce a useful blueprint for software engineers and framework developers.

This is not to say that this thesis brings merely practical insight – the thesis also improves the state-of-the-art in concurrent and parallel computing. Researchers in the field of parallel computing should find the contributions on the subject of persistent and concurrent data structures, scheduling algorithms and data parallelism very valuable. We hope that the ideas and implementations, analyses and proofs, as well as some important fundamental discoveries in this thesis will serve as a useful foundation for driving future research.

1.6 Preliminaries

The code examples in this thesis are written in Scala. In Chapters 4 and 5 we frequently use pseudocode when presenting lock-free concurrent algorithms. This decision is made to simplify the understanding of these algorithms and to clearly highlight their important components, such as the linearization points. Nevertheless, reading the thesis requires some knowledge of Scala, but the differences with most other mainstream languages are only syntactic. In fact, there are many similarities between Scala and other languages, so readers with different backgrounds should find the code understandable.

In Scala, variables are declared with the keyword `var` much like in JavaScript or C#, and their type is inferred from the initialization expression. Methods are defined with the keyword `def` like in Python or Ruby, and type ascriptions come after a `:` sign. First-class functions are written as an argument list followed by their body after the `=>` arrow, similar to Java 8 lambdas. Like the `final` modifier in Java, keyword `val` denotes fields and local variables that cannot be reassigned. The keyword `trait` denotes an `interface`, but can also have concrete members – in fact, `implements` is called `with` in Scala. The keyword `object` denotes a placeholder for `static` methods belonging to some type. Generic types are enclosed in `[]` brackets, similar to the `<>` notation in other languages. The type `Unit` is the analogue of the `void` type in C or Java. Statements in the body of a `class` or `trait` are considered parts of its primary constructor. Finally, suffixing an object with an expression list in parenthesis syntactically rewrites to calling its `apply` method – this is similar to `operator()` overloading in C++. After this brief overview of Scala, we feel that readers with a good Java, Scala or C# background will have no problem reading this thesis.

Some basic understanding of concurrent programming and multithreading is also welcome. We feel that a basic operating systems course or knowledge about concurrency in Java should be sufficient to understand the algorithms in this thesis. For readers seeking more insight into these topics, we strongly recommend the books *Art of Multiprocessor Programming* by Herlihy and Shavit [Herlihy and Shavit(2008)], and *Java Concurrency in Practice* [Goetz et al.(2006)Goetz, Bloch, Bowbeer, Lea, Holmes, and Peierls].

1.7 Our Contributions

This thesis studies how to efficiently implement a nested data-parallel programming model for managed runtimes running on top of shared-memory multiprocessor systems in a generic way for a broad range of general-purpose data structures and data-parallel operations. The contributions brought forth in this thesis are the following:

- Generic parallel collection framework – a systematic data-parallel collections framework design, along with an efficient implementation for modern multiprocessors, described in Chapter 2. The generic data-parallel collection framework allows defining new collections and operations orthogonally, and is thus extensible with custom operations. We describe a series of data structures especially designed for data-parallel computations accompanied by their implementations.

- Conc-trees – efficient sequence implementations such as conc-lists, conc-ropes, conqueues and various Conc-tree buffers in Chapter 3. We show how Conc-trees can support efficient $O(1)$ deque operations, $O(\log n)$ concatenation and traversal in optimal time. We analyze the running time of their operations and provide their implementations.

- Flow pools – an abstraction for deterministic reactive data-parallelism, and an efficient lock-free concurrent queue implementation used to implement this abstraction described in Chapter 4.

- Ctries – a lock-free concurrent map implementation and a novel approach to executing linearizable, lock-free snapshots on concurrent data structures proposed in Chapter 4. We use the snapshots to provide safe data-parallel operations on concurrent data structures. A novel insight in this chapter is that the use of *lazyness* with lock-free snapshots allows copying to be more easily parallelized.

- Work-stealing tree scheduling – the work-stealing tree data structure and the work-stealing tree scheduler, a novel lock-free scheduling algorithm for data-parallel computations that is efficient for both fine-grained uniform and severely irregular data-parallel workloads described in Chapter 5.

- We evaluate the algorithms and data structures presented in this thesis experimentally in Chapter 6.

- We present correctness proofs for several data structures from this thesis in the appendices. We feel that these proofs will be insightful both to researchers and practitioners. In particular, both groups of readers may eventually decide to implement their own *lock-free* data structures – reasoning about their correctness is much more challenging than reasoning about single-threaded data structures. It is therefore recommended to use our proofs as templates for constructing your own.

In the next chapter, we start by introducing the design of the generic parallel collection framework, which serves as a foundation for the remainder of this thesis.

2 Generic Data Parallelism

Data-parallel operations can be implemented for a wide range of different data structures and operations – since the Scala standard library collections framework [Odersky(2009)] consists of dozens of bulk operations, it would be very hard to write and maintain a framework that reimplements those operations for every data structure. In this chapter we study a generic approach to implementing data-parallel collections in a managed runtime. Here, generic means that data-parallel operations can be implemented only once and then used for a wide range of data structures granted that they meet certain requirements. As we will see, this greatly simplifies the implementation and maintenance of a data-parallel framework.

2.1 Generic Data Parallelism

Data elements might be stored inside arrays, binary search trees or hash-tables – they should all support the same data-parallel operations. Data-parallel operations should also be generic in the type of elements contained in the data structure. If a client requests to find the largest element of a collection, the same operation should work with string, number or file handle elements. Finally, most data-parallel operations are parametrized by the client at a particular invocation site. In the example with the largest element, the operation is parametrized by the function that compares string lengths, two numbers

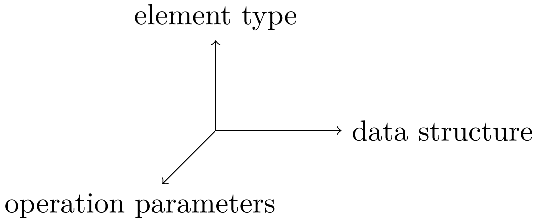


Figure 2.1: Data-Parallel Operation Genericity

or the space consumption of a file. We illustrate these relationships as three orthogonal axes in Figure 2.1.

We rely on several assumptions about data structures in this chapter. While these assumptions might seem natural, they are essential for the parallelization approach we propose. In the later chapters we will drop some of these constraints on specific data structures and show several different approaches to data-parallelism.

First of all, in this chapter we assume that data-parallel operations execute in a *bulk-synchronous manner*. This means that the data-parallel operation caller blocks execution, the data-parallel operation then starts and completes, and then the caller resumes execution. Although this constraint can easily be dropped by applying `Future`s from Section 4.1.1 to data-parallel operations in this chapter, it has a nice consequence that the caller cannot modify the data structure while the data-parallel operation is in progress.

Second, we assume *element presence* – elements we want to execute the operation on are already present in the data structure before the operation starts. Some data structures are designed for streaming and are supply-driven, so their elements can be become available as they arrive. We do not consider them in this chapter.

Finally, we assume *quiescence* during the execution of a data-parallel operation lifetime. This means that the data structure is not modified during the entire execution of the data-parallel operation. These modifications include adding new elements to the data structure, and removing and modifying existing elements. Note that quiescence implies element presence, but the presence of elements when the operation starts does not imply quiescence – the distinction will be crucial in Section 4.2 when we apply data-parallelism to concurrent data structures.

2.2 Data Parallel Operation Example

In this section we show an example of several concrete data-parallel operations along with their implementation. We start with a simple operation `reduce` that applies a user-specified binary associative operator `op` to collapse all the elements of the collection into a single value. The sequential implementation of the `reduce` method has the following signature and implementation in the Scala standard collection library:

```
def reduce[U >: T](op: (U, U) => U): U = {
  val it = iterator
  var sum = it.next()
  while (it.hasNext) sum = op(sum, it.next())
  sum
}
```

This implementation is generic across all the axes mentioned in the previous section – the creation of the `iterator` makes it data-structure-agnostic, the type of the elements in the data structure is represented with an abstract type `T` and the specific operator `op` is specified at the callsite – `op` could be doing summation, concatenation or returning the bigger of the two elements according to some ordering. In fact, given that `op` is a pure function, i.e. does not have any side-effects, we can even execute `reduce` in parallel. Lets assume that we can call a method `task` that takes a block of code and returns a `Task` object that can execute that block of code asynchronously. Any `Task` object can be executed asynchronously by calling `fork` – upon calling `fork` another processor executes the task's block of code and stores the value of that block into the task object. The caller can block and wait for the result produced by the asynchronous `Task` computation by calling its `join` method.

```
def parReduce[U >: T](op: (U, U) => U): U = {
  val subiterators = split(iterator)
  val tasks = for (it <- subiterators) yield { it =>
    task {
      var sum = it.next()
      while (it.hasNext) sum = op(sum, it.next())
      sum
    }
  }
  for (t <- tasks) t.fork()
  val results = for (t <- tasks) yield t.join()
  results.reduce(op)
}
```

The `parReduce` method starts by calling a `split` method on the iterator. Given a freshly created `iterator` this method returns a set of iterators traversing the subsets of the original iterator. We defer the details of how `split` is implemented until Section 2.3. Each of those subset iterators `it` is then mapped into a new `Task` object that traverses the corresponding iterator `it` and computes the `sum`. The caller thread `fork`s those task objects and then calls `join` on each of them – the task objects are in this way mapped into the list of `results`. The `results` computed by different task objects are then `reduce`d sequentially by the caller thread.

The simple implementation presented above has certain limitations. In particular, it is unclear how to implement `split` efficiently on iterators for arbitrary data structures. Here, efficiently means that the `split` takes $O(P \log n)$ time where n is the number of elements that the iterator traverses and P is the number of workers. Then, the `split` may return a certain number of subset iterators that is lower than the number of available processors. A *data-parallel scheduler* must ensure that all the available processors are assigned useful work whenever possible. Orthogonally to efficient scheduling, processors must execute the workload assigned to them as efficient as possible. Iterators, function objects and genericity in the collection element type represent an abstraction cost that can slow down an operation dramatically, but can be overcome with proper compile-time optimisations. Finally, certain data structures do not have an efficient `split` implementation and must be parallelized in some other fashion.

```
trait Iterator[T] {
  def next(): T
  def hasNext: Boolean
}

trait Splitter[T] extends Iterator[T] {
  def split: (Splitter[T], Splitter[T])
}

trait PreciseSplitter[T] extends Splitter[T] {
  def psplit(sizes: Seq[Int]): Seq[PreciseSplitter[T]]
}
```

Figure 2.2: Splitter Interface

We address those issues in the remainder of this chapter.

2.3 Splitters

For the benefit of easy extension to new parallel collection classes and easier maintenance we want to define most operations in terms of a few abstract methods. The approach taken by Scala standard library sequential collections is to use an abstract `foreach` method and iterators. Due to their sequential nature, both are not applicable to data-parallel operations – they only produce one element at a time and do not allow access to subsequent elements before prior elements are traversed. In addition to element traversal, we need a splitting operation that returns a non trivial partition of the elements of the collection. The overhead of splitting the collection should be as small as possible – this influences the choice of the underlying data structure for parallelization.

We therefore define a new abstraction called a *splitter*. Splitters are an extension of iterators with standard methods such as `next` and `hasNext` that are used to traverse a dataset. Splitters have a method `split` that returns two child splitters, which iterate over disjunct subsets of the collection. The original iterator becomes invalidated after calling `split` and its methods must no longer be called. The `Splitter` definition is shown in Figure 2.2.

Method `split` returns a sequence of splitters such that the union of the elements they iterate over contains all the elements remaining in the original splitter. These subsets are disjoint. Additionally, these subsets may be empty if there is 1 or less elements remaining in the original splitter [1]. For the purposes of achieving decent load-balancing, data-parallel schedulers may assume that the datasets returned by the `split` method are in most cases roughly equal in size.

The `split` operation is allowed to execute in $O(P \log n)$ time, where P is the number of

[1] They may additionally be empty if recursive splitting returns two non-empty splitters after a finite number of splits. This fineprint exists to ensure termination during scheduling, while simplifying the implementation of certain splitters.

workers and n is the size of the underlying data structure. Most common data structures can be split in $O(1)$ time easily, as we will see.

A *precise splitter* is more specific splitter `PreciseSplitter` that inherits `Splitter` and allows splitting the elements into subsets of arbitrary sizes. This splitter extension is used by parallel sequences and is required to implement sequence operations like zipping which needs to split multiple data structures at once.

2.3.1 Flat Splitters

Flat data structures are data structures in which elements are contained in a contiguous block of memory. This block of memory need not be completely filled with elements. Such data structures are particularly amenable to parallelization, since a contiguous block of memory can easily be divided into smaller blocks.

Array and Range Splitters

An array is a ubiquitous data structure found in almost all general purpose languages. In the standard Scala collections framework it is one of the basic mutable sequence collections. Arrays have efficient, constant time element retrieval by index. This makes them particularly suitable for parallelization. Similar data structures like integer ranges and various array-based sequences like Scala `ArrayBuffer`s implement splitters in the same manner.

An `ArraySplitter` implementation is shown in Figure 2.3. This splitter contains a reference to the array, and two indices for the iteration bounds. The implementation is trivial – method `split` divides the iteration range in two parts, the second splitter starting where the first ends. This makes `split` an $O(1)$ method.

Hash Table Splitters

Standard Scala collection library has four different collections implemented in terms of hash tables – mutable `HashMap`s and `HashSet`s, as well as their linked variants that guarantee traversal in the order that the elements were inserted. Their underlying hash table is an array with both empty and non-empty entries, but the elements are distributed uniformly in the array. Every element in this array is mapped to a specific index with a *hashing function*. In some cases multiple elements map to the same index – we call this a collision. Hash tables can be implemented with *closed addressing*, meaning that collisions are resolved by storing a data structure with potentially several elements at the same index, or *open addressing*, meaning that collisions are resolved by putting one of the collided elements at some other index in the hash table

```
class FlatSplitter[T] extends Splitter[T] {
  def newSplitter(lo: Int, hi: Int): FlatSplitter[T]
  def split = Seq(
    newSplitter(i, (i + until) / 2),
    newSplitter((i + until) / 2, until))
}

class ArraySplitter[T](val a: Array[T], var i: Int, val until: Int)
extends Splitter[T] {
  def hasNext = i < until
  def next() =  i += 1; a(i - 1)
  def newSplitter(lo: Int, hi: Int) = new ArraySplitter(a, lo, hi)
}

class FlatHashSplitter[T](val a: Array[T], var i: Int, val until: Int)
extends Splitter[T] {
  private def advance() = while (a(i) == null) i += 1
  def hasNext = i < until
  def next() =  val r = a(i); i += 1; advance(); r
  def newSplitter(lo: Int, hi: Int) = new FlatHashSplitter(a, lo, hi)
}
```

Figure 2.3: Flat Splitter Implementations

[Cormen et al.(2001)Cormen, Leiserson, Rivest, and Stein].

Scala mutable sets are implemented with open addressing, whereas mutable maps and linked variants of maps and sets use closed addressing. Figure 2.3 shows a splitter implementation `FlatHashSplitter` for a hash set implemented with open addressing. The closed addressing implementation is slightly more verbose, but similar. We note that data-parallel operations on linked maps and sets cannot maintain insertion-order traversal. The reason is that the simple, yet efficient linked list approach to maintaining insertion order does not allow parallel traversal.

2.3.2 Tree Splitters

Previous section showed splitters for data structures composed of a single contiguous part of memory. Trees are composed of multiple objects such that there is a special `root` node, and that every other object in the tree can be reached in exactly one way by following some path of pointers from the `root` to that node. Trees that we will consider need not be binary – nodes can have any fixed upper bound on the number of children. In this section we investigate splitters for trees that are balanced – any two paths between the root and a node differ in length by a constant factor for any tree, irrespective of the number of elements stored in that tree. One useful consequence of this constraint is that the depth of balanced trees is bound by $O(\log n)$, where n is the number of elements stored in the tree.

Trees that are not balanced usually cannot be efficiently parallelized using splitters. We note that the majority of tree-based data structures are balanced, as this property ensures good asymptotic bounds on useful data-structure operations.

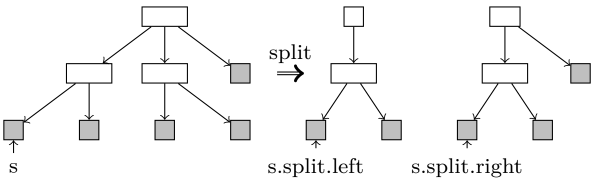


Figure 2.4: Hash Trie Splitting

With this in mind, we will divide the trees in several categories. *Combining trees* are trees in which every node contains some metric about the corresponding subtree. *Size-combining trees* are trees in which every node contains the size of the subtree – splitters are easily implementable for such trees. Updates for persistent trees usually involve updating the path from the root to the updated node, so augmenting such trees with size information is often easy. *Two-way trees* are trees in which every node contains a special pointer to its parent[2], such that this path eventually leads to the `root` of the tree, which has no parent pointer. The third class are *one-way trees*, in which nodes do not hold a pointer to the parent tree. The splitters in the last two groups are more complex to implement.

Hash Trie Splitters

Scala standard library implements persistent hash tables in terms of hash tries with a high branching factor (up to 32-way nodes) [Bagwell(2001)]. For a typical 32-bit hashcode space used on the JVM, these tries have a maximum depth of 6 nodes, resulting in more efficient update and lookup operations.

Hash tries are size-combining trees – every node contains the total number of elements contained in the corresponding subtree. Nodes can be either internal nodes or leaves. Every internal node contains an array of pointers to its children nodes, but it does not contain elements. Elements are contained in leaves. Each leaf contains a single element (or a single key-value pair in case of maps). Additionally, internal nodes compress empty pointer entries using a special bitmap flag. Memory footprint is thus reduced compared to binary tries. The hash trie data structure internals are discussed in more detail in Section 4.2.2.

Hash trie splitters conceptually divide the hash trie into several smaller hash tries, as illustrated in Figure 2.4. Each splitter resulting from calling the `split` method on a splitter `s` holds a reference to one of the smaller hash tries. Although such splitting can be done in $O(\log n)$ time, the implementation can avoid physically creating smaller hash tries by maintaining iteration progress information in each of the splitters, as we show in Figure 2.5.

[2] For the purposes of the tree data structure definition we ignore the existence of this pointer.

```
class HashTrieSplitter[T]
  (val root: Node[T], var progress: Int, val until: Int)
extends Splitter[T] {
  private var depth = -1
  private var index = new Array[Int](6)
  private var nodes = new Array[Node[T]](6)
  private def init() {
    nodes(0) = root
    index(0) = 0
    depth = 0
    var tot = 0
    def descend() {
      while (tot + nodes(depth).array(index(depth)).size <= progress)
        index(depth) += 1
      val currnode = nodes(depth).array(index(depth))
      if (!currnode.isLeaf) {
        push(currnode)
        descend()
      }
    }
    descend()
  }
  init()
  private def push(n: Node[T]) {
    depth += 1
    index(depth) = -1
    nodes(depth) = n
  }
  private def pop() {
    nodes(depth) = null
    depth -= 1
  }
  private def advance() {
    index(depth) += 1
    if (index(depth) < nodes(depth).array.length) {
      val currnode = nodes(depth).array(index(depth))
      if (!currnode.isLeaf) {
        push(currnode)
        advance()
      }
    } else {
      pop()
      advance()
    }
  }
  def next(): T = if (hasNext) {
    val leaf = nodes(depth).array(index(depth))
    advance()
    progres += 1
    leaf.asLeaf.element
  } else throw new NoSuchElementException
  def hasNext: Boolean = progress < until
  def split = {
    new HashTrieSplitter(root, progress, (progress + until) / 2),
    new HashTrieSplitter(root, (progress + until) / 2, until))
  }
}
```

Figure 2.5: Hash Trie Splitter Implementation

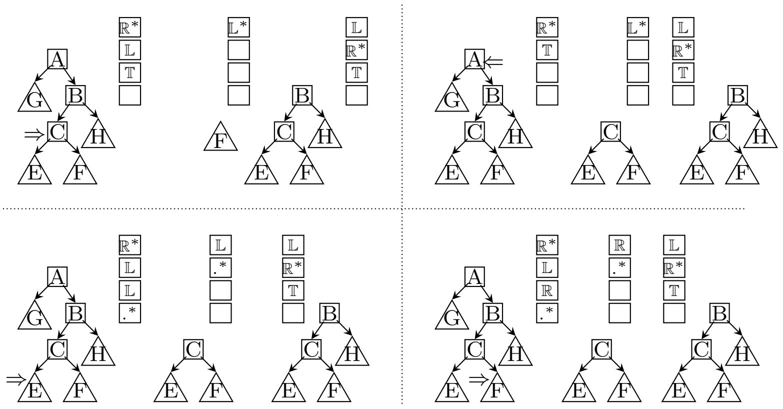


Figure 2.6: `BinaryTreeSplitter` splitting rules

The implementation in Figure 2.5 maintains the iteration state using an integer field `progress` and two stacks of values. The stacks maintain the path from the root of the hash trie to the currently traversed leaf node. Upon creation, the stacks are set to point to the element at position `progress` in the left-to-right trie traversal. This is done by the method `init` in line 40, which skips subtrees as long as the total number of skipped elements is less than the specified `progress`, and then recursively descends into one of the children. In line 46 size information is used to efficiently initialize a splitter.

Traversal is no different than traversal in standard iterators. Every time `next` is called, the current leaf is read in line 80 and then the position is advanced to the next leaf node. Method `advance` has to check if there are more leaf nodes in the current inner node in line 68. If there are no more leaf nodes, the current inner node has to be popped of the stack in line 75. If there are more leaf nodes, the `advance` method checks if the next leaf node is a leaf. Non-leaf nodes need to be recursively pushed to the stack in line 71.

Method `split` divides the range from `progress` to `until` into two subranges of roughly equal size, and then creates two new hash trie splitters whose `init` method is responsible for setting their state correctly.

Scala Vector Splitters

Scala `Vector` [Bagwell and Rompf(2011)] is an immutable sequence implementation. Similar to the hash tries from the previous section, it has up to 32-way branching factor in the inner nodes and stores elements in the leaf nodes. Scala `Vectors` are also size-combining trees, since the size of each subtree can be implicitly computed. This means that Scala `Vector` splitters can be implemented analogous to hash trie splitters.

Binary Search Tree Splitters

The previous two tree data structures were size-combining trees. We now turn our attention to splitters for one-way and two-way trees. Note that any technique of splitting a one-way tree is applicable to a two-way tree. This is because any splitter for a two-way tree can simply ignore the parent pointer and treat the tree as if it were a one-way tree. Therefore, in this section we examine the techniques for implementing one-way tree splitters. In particular, we will assume that the trees are one-way binary balanced trees for the purposes of this section, but the techniques presented here can be applied to one-way n-ary balanced trees as well.

In the Scala standard library `SortedSet`s and `SortedMap`s are implemented in terms of one-way red-black trees which are binary balanced trees.

Similar to hash tries shown in Section 2.3.2, we note that the state of the iteration for binary trees can be encoded using a stack. Unlike the size-combining trees, one-way tree splitters cannot split the tree by maintaining a `progress` field and then initializing new splitters using this information. The absence of this information prevents us from skipping subtrees as in line 46 of `HashTrieSplitter`s. Without skipping the subtrees the entire tree needs to be traversed and this changes the complexity of `split` from $O(\log n)$ to $O(n)$. The only traversal information that `BinaryTreeSplitter`s maintain is the stack of nodes on the path from the root to the current leaf, so the `split` method must use that.

A binary tree splitter state can be logically represented with a single stack of decisions of whether to go left ($\mathbb{L}$) or right ($\mathbb{R}$) at an inner node. As most binary search trees store elements in inner nodes, a binary tree splitter state must encode the fact that a specific inner node has been traversed by ending the stack with a symbol $\mathbb{T}$.

We do not show the complete splitting pseudocode, but show the important classes of different states the stealer can be in, in Figure 2.6. We encode the state of the stack with a regular expression of stack symbols. For example, the regular expression $\mathbb{R}^*\mathbb{L}\mathbb{T}$ means that the stack contains a sequence of right turns followed by a single left turn and then the decision $\mathbb{T}$ to traverse a single node. A stealer in such a state should be split into two stealers with states $\mathbb{L}^*$ on the subtree F and $\mathbb{L}\mathbb{R}^*\mathbb{T}$ on the subtree B, as shown in Figure 2.6. This splitter implementation does not require allocating any tree nodes as the newly constructed splitters take subtrees of the existing tree as arguments.

Note that splitting trees in this way might not always yield splitters over subsets that have approximately the same sizes. The sizes of the subtrees assigned to child splitters differ up to a constant factor – this follows from the fact that the trees (and their subtrees) are balanced. However, the left child splitter may have already traversed some of the elements in its tree, so its subset of elements may be arbitrarily smaller. We have found that this is not an issue in practice, as there are many splits and the data-parallel

```
trait Builder[T, That] {
  def +=(elem: T): Unit
  def result: That
}

trait Combiner[T, That, Repr] extends Builder[T, That] {
  def combine(other: Repr): Combiner[T, That]
}
```

Figure 2.7: Combiner Interface

```
class Map[S, That](f: T => S, s: Splitter[T])
extends Batch[Combiner[S, That]] {
  var result: Combiner[S, That] = newCombiner
  def split = s.split.map(p => new Map[S](f, p))
  def leafTask() {
    while (s.hasNext) cb += f(s.next)
  }
  def merge(that: Map[S, That]) {
    cb = cb combine that.cb
  }
}
```

Figure 2.8: `Map` Task Implementation

schedulers are designed to deal with unbalanced workloads anyway.

2.4 Combiners

While splitters allow assigning collection subsets to different processors, certain operations return collections as their result (e.g. `map`). Parts of the collection produced by different workers must be combined together into the final collection. To allow implementing such data-parallel operations generically for arbitrary operations we introduce an abstraction called a *combiner*.

In Figure 2.7, type parameter `T` is the element type, and `That` is the resulting collection type. Each parallel collection provides a specific combiner, just as regular Scala collections provide builders. The method `combine` takes another combiner and produces a combiner that contains the union of their elements. Combining may occur more than once during a parallel operation, so this method should ideally have complexity $O(1)$ and no more than $O(P \log n)$, where n is the number of elements in the combiners and P is the number of processors.

Parallel operations are implemented within task objects, as discussed in Section 2.7. These tasks correspond to those described in the previous section. Each task defines `split` and `merge` methods. To illustrate the correspondence between task objects, splitters and combiners, we give an implementation of the task for the parallel `map` operation in Figure 2.8.

The `Map` task is given a mapping function `f` of type `T => S` and a splitter `s`. These tasks must be split into smaller chunks to achieve better load balancing, so each task has a `split` method, which it typically implements by calling `split` on the splitter and mapping each of the subsplitters into subtasks, in this case new `Map` objects. At some point this splitting ends and the data-parallel scheduler decides to call the method `leafTask`, mapping the elements and adding them into the combiner for that task. Results from different workers are then merged hierarchically using the `merge` method, which in this case combines the combiners produced by two different tasks. Once the computation reaches the root of the tree, a method `result` is called on its combiner `cb` to obtain the resulting collection.

The challenging part of implementing a combiner is its `combine` method. There are no predefined recipes on how to implement a combiner for any given data structure. This depends on the data-structure at hand, and usually requires a bit of ingenuity.

Some data structures have efficient implementations (usually logarithmic) of these operations. If the collection at hand is backed by such a data-structure, its combiner can be the collection itself. Finger trees, ropes and binomial heaps implement efficient concatentation or merging operations, so they are particularly suitable. We say that these data structures have *mergeable combiners*.

Another approach, suitable for parallel arrays and parallel hash tables, assumes that the elements can be efficiently stored into an intermediate representation from which the final data structure can be created. This intermediate representation must implement an efficient merge operation, and must support efficient parallel traversal. In this approach several intermediate data structures are produced and merged in the first step, and the final data structure is constructed in the second step. We refer to such combiners as *two-step combiners*.

While the last two approaches actually do not require any synchronization primitives in the data-structure itself, they assume that it can be constructed concurrently in a way such that two different processors never modify the same memory location. There exists a large number of concurrent data-structures that can be modified safely by multiple processors -– concurrent skip lists, concurrent hash tables, split-ordered lists, concurrent avl trees, to name a few. They can be used to create *concurrent combiners*. An important consideration is that the concurrent data-structure has a horizontally scalable insertion method. For concurrent parallel collections the combiner can be the collection itself, and a single combiner instance is shared between all the processors performing a parallel operation.

2.4.1 Mergeable Combiners

Combiners that implement an $O(P \log n)$ worst-case `combine` and `result` operations are called mergeable combiners. Mergeable combiners are a desireable way to implement combiners, both in terms of simplicity and in terms of performance. An existing merge or concatenation data structure operation is used to implement `combine`. The `result` operation simply returns the underlying combiner data structure, optionally doing $O(P \log n)$ sequential postprocessing work. Unfortunately, most data structures do not have an efficient merge or concatenate operation. In fact, linked list data structures, skip lists, catenable random-access sequence implementations and certain priority queue implementations comprise the host of known mergeable data structures. Out of those, linked lists are not suitable for subsequent parallelization, as efficient linked list splitters do not exist. We show several mergeable combiner implementations in Chapter 3, where we introduce the Conc-tree data structure.

2.4.2 Two-Step Combiners

Most standard library collections do not come with an efficient merge or concatenation operation. In particular, mutable sequences implemented with arrays or queues, mutable maps and sets implemented with hash tables or binary search trees, or immutable maps and sets implemented with hash tries do not have mergeable combiners. In most cases these data structures have two-step combiners. These combiners use an intermediate data structure to store results produced by different processors. The intermediate data structure is used to produce the final data structure.

There are several constraints on these intermediate data structures. First, element addition must be $O(\log n)$, and preferably $O(1)$ with good constant factors. Second, they must support a $O(P \log n)$ time merge operation. Finally, they must support parallel traversal in a way that subsets of elements with a high spatial locality in the final data structure can be traversed by a particular processor efficiently. The final constraint ensures that the final data structure can be constructed with minimal synchronization costs. We visit several examples of these intermediate data structures in the following subsections and later in Sections 3.4.1 and 3.4.2. While the intermediate data structures support efficient merging, they typically do not support efficient operations of the final data structure.

As their name implies, two-step combiners are used in two steps. In the first step different processors independently produce combiners with some results and then concatenate them together. In the second step this concatenated intermediate data structure is independently traversed to create the final data structure. We study several concrete examples next.

Array Combiners

Given two arrays, there is no sublinear time operation that produces their concatenation. This means that the intermediate combiner data structure cannot be an array. Instead,

```
class ArrayCombiner[T](val chunks: ArrayBuffer[ArrayBuffer[T]], var size: Int)
extends Combiner[T, Array[T], ArrayCombiner[T]] {
  def +=(elem: T) {
    size += 1
    chunks.last += elem
  }
  def combine(that: ArrayCombiner[T]) = {
    new ArrayCombiner(chunks ++= that.chunks, size + that.size)
  }
  def result = {
    val array = new Array[T](size)
    val t = new ToArrayTask(chunks, array)
    t.fork()
    t.join()
    array
  }
}
```

Figure 2.9: Array Combiner Implementation

the basic array combiner implementation uses an array of arrays of elements. Each nested array carries elements produced by a certain worker, added to the combiner using the `+=` method. We call these inner arrays *chunks*. Chunks are growable – once they are completely filled with elements, a larger contiguous memory area is allocated, and the existing elements are copied into it. The outer array is also growable and holds a list of chunks.

The implementation is shown in Figure 2.9. In the Scala standard library, the growable array collection is called `ArrayBuffer`. Adding to the combiner with the `+=` method simply adds the element to the last chunk. Combining works by concatenating the list of chunks. Assuming there are always as many chunks as there are different workers, `combine` is an $O(P)$ operation. Once all the combiners are merged, the `result` method is called – the final array size is known at this point, so `result` allocates the array, and executes the `ToArray` task that copies the array chunks into the target array (we omit the complete code here). Copying proceeds without synchronization as different chunks correspond to non-overlapping contiguous parts of the array.

Note that certain data-parallel operations, which create parallel arrays and know their sizes in advance, (e.g. `map`) can allocate the final array at the beginning of the operation, and work on it directly. These operations are special cases that do not need to use combiners.

While this implementation works well in many cases, it has several disadvantages. First, every element is on average copied twice into its chunk. Then, this representation wastes half the memory in the worst case, as every growable array can be half-empty. Finally, this implementation assumes that the number of chunks is $O(P)$. Certain data-parallel schedulers that use task-based parallelism can create a number of tasks much larger than the number of processors, allowing the `chunks` array to grow beyond $O(P)$ and slowing down the `combine` operation. We show a more efficient array combiner implementation

```
class HashSetCombiner[T](val ttk: Int, val buckets: Array[UnrolledList[T]])
extends Combiner[T, HashSet[T], HashSetCombiner[T]] {
  def +=(elem: T) {
    val hashcode = elem.hashCode
    buckets(hashcode & (ttk - 1)) += elem
  }
  def combine(that: HashSetCombiner[T]) = {
    for (i <- 0 until ttk) buckets(i) concat that.buckets(i)
    new HashSetCombiner(ttk, buckets)
  }
  def result = {
    val numelems = buckets.foldLeft(0)(_ + _.size)
    val sz = nextPowerOfTwo(numelems / loadFactor)
    val array = new Array[T](sz)
    val t = new ToTable(buckets, array)
    t.fork()
    t.join()
    new HashSet(array, sz)
  }
}
```

Figure 2.10: `HashSet` Combiner Implementation

in Chapter 3.

Hash Table Combiners

If each hash table combiner were to maintain a hash table of its own, combining would require traversing both hash tables and merging corresponding buckets together, resulting in an $O(n)$ combine. Again, hash tables are inadequate as an intermediate data structure [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky].

Recall, from the beginning of this section, that the intermediate data structure needs to assign different subsets of elements to different processors. In addition, we want each of the subsets to occupy a contiguous chunk of memory, to avoid unnecessary synchronization and false sharing [Herlihy and Shavit(2008)]. To achieve this, we partition the space of elements according to their hashcode prefixes so that the elements end up in different contiguous blocks in the final hash table. This partition is independent of the size of the resulting hash table. The intermediate data structure serves as a set of buckets for this partition. The blocks in the resulting hash table can then be filled in parallel by multiple processors that traverse different buckets in the intermediate data structure. We describe how to do this partition next.

Each combiner keeps an array of 2^k buckets, where 2^k is a constant bigger than the number of processors. Experimental results suggest that 2^k should be up to an order of magnitude bigger than the number of processors to ensure good load balancing, and $k = 5$ works well for up to 8 processors. Each bucket is an unrolled linked list. In general, unrolled lists have the downside that indexing an element in the middle has complexity $O(n/m)$ where n is the number of elements in the list and m is the chunk size, but this is not a problem in our case since elements are never indexed, only added at the end and

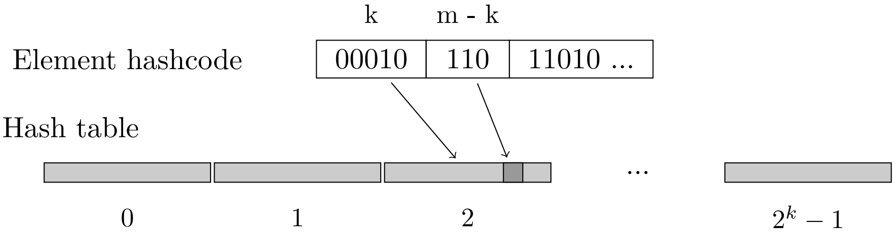


Figure 2.11: Hash Code Mapping

then traversed once. We add an element by computing its hashcode, and then inserting it into a bucket indexed by the k first bits of the hashcode. Adding an element to an unrolled linked list is fast – in most cases it amounts to incrementing an index and storing an element into an array, and only occasionally allocating a new node. Combining works by `concat`enating each unrolled linked list in $O(1)$ time – we choose 2^k such that there are $O(P)$ unrolled linked list, making `combine` an $O(P)$ operation.

Method `result` computes the total number of elements by adding the sizes of the buckets together. The size of the hash table is then computed by dividing the number of elements with the load factor and rounding it to the smallest larger power of 2. The array for the hash table is allocated and the `ToTable` task is forked, which can be split in up to 2^k subtasks. This task copies the elements from different buckets into the hash table. Assume hash table size is $sz = 2^m$. We obtain the position in the hash table by taking the first m bits from the hashcode of the element. The first k bits denote the index of the block within the hash table, and the remaining $m - k$ bits denote the position within that block (see example in figure 2.11). Since all the elements in a bucket have identical first k bits, they are always written into the same hash table block. We show the implementation for `HashSet`s in Figure 2.10.

Note that, as described so far, the hash code of each key is computed twice. The hash code is first computed before inserting the key into the combiner. It is computed again when adding the keys into the hash table. If computing the hash code is the dominating factor in the parallel operation, we can avoid computing the hash code twice by storing it into the combiner unrolled linked lists, along with the key and the value.

A nice property of hash tables with closed addressing is that the elements are always written into their respective blocks. With open addressing, it is possible that some of the elements *spill* out of the block. The `ToTable` task object records all such elements, and tries to reinsert them into the next block when merging with the next task. This results in the average number of spills equal to the average collision lengths in the hash table and is on average constant, so the extra work of handling spilled elements is bound by $O(P)$ [Sujeeth(2013)] [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky].

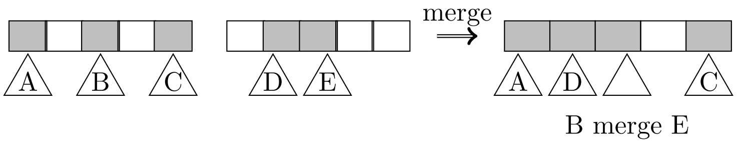


Figure 2.12: Hash Trie Combining

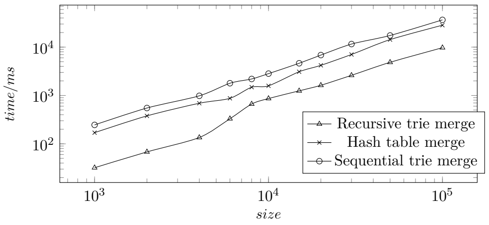


Figure 2.13: Recursive Trie Merge vs. Sequential Construction

Hash Trie Combiners

Hash trie combiners can be implemented to contain hash tries as the intermediate data structure. Merging the hash tries is illustrated in Figure 2.12. For simplicity, the hash trie root nodes are shown to contain only five entries. The elements in the root hash table are copied from either one hash table or the other, unless there is a collision as in the case of subtries B and E. Subtries that collide are recursively merged and the result is put in the root hash table of the resulting hash trie. This technique turns out to be more efficient than sequentially building a hash trie.

We compare the performance of recursively merging two existing tries against merging two hash tables against sequentially constructing a new merged trie in Figure 2.13 – recursive merging results in better performance than either sequential hash trie or hash table merging. The merging could also be done in parallel. Whenever two subtries collide, we could fork a new task to merge the colliding tries. Since the elements in the two colliding tries all share the common hashcode prefix, they will all end up in the same subtrie – the subtrie merge can proceed independently of merging the rest of the trie. This approach is applicable only if the subtries merged in a different task are large enough.

Parallel recursive merging has $O(n/P)$ complexity in the worst case. In a typical parallel operation instance, the `combine` method is invoked more than once (see Figure 2.17). Hash tries are thus not efficient intermediate data structures.

The intermediate data structure is instead akin to the one used by hash table combiners. It consists of an array of 2^k unrolled lists, each holding elements with the same k bit hashcode prefix (where $k = 5$), as before.

Combiners implement the `combine` method by simply going through all the buckets, and concatenating the unrolled linked lists that represent the buckets, which is an $O(P)$ operation. Once the root combiner is produced, the resulting hash trie is constructed in parallel in `result` – each processor takes a bucket and constructs the subtrie sequentially, then stores it in the root array. Again, if computing the hash code is expensive, we can cache the hash code of each key in the bucket. Another advantage of this approach is that each of the subtries is on average one level less deep, so a processor working on a subtrie does less work when adding each element.

Binary Search Tree Combiners

In most languages and runtimes, ordered sets are implemented with AVL or red-black trees. To the best of our knowledge, there is no algorithm for efficient (e.g. asymptotic $O(\log n)$ time) AVL or red-black tree union operation. This means that we cannot implement directly mergeable binary search tree combiners, but have to use the multiple-step evaluation approach. Binary search tree combiners rely on bucketing and merging non-overlapping subsets, as we describe in the following.

Note that, while AVL or red-black trees can be merged in $O(\min(n, m) \cdot \log(\max(n, m)))$ time, where n and m are their respective sizes, two such binary search trees can be merged more efficiently if all the elements in the first tree are smaller than the elements in the second tree. One way of doing this is the following. First, we determine the higher tree. Without the loss of generality, let's assume that the second (right) tree T_2 is higher. First, we need to extract the rightmost node from the first (left) tree T_1 – call it X. After removing X from T_1 in $O(\log n)$ time, we are left with the T_1'. We need to find a node Y on the leftmost path in the right tree T_2, whose height differs from the height of T_1' by at most 1 – we can do this in $O(\log m)$ time. The tree T_1', node X and the subtree at the node Y are non-overlapping – they can be linked into the tree T_{1Y} in $O(1)$ time. We replace the subtree at Y in the tree T_2 with T_{1Y}, and rebalance all the nodes on the path to the root – this is a $O(\log m)$ operation.

Combiners produced by separate processors are not guaranteed to contain non-overlapping ranges (in fact, they rarely do), so we need to use a bucketing technique, similar to the one used in hash tables and hash tries, to separate elements into non-overlapping subsets. Recall that the buckets were previously conveniently induced by bits in the hashing function. A good hashing function ensured that separate buckets have the same expected number of elements, regardless of the element set contained in the hash table. Since binary search trees can contain elements from arbitrary ordered sets, choosing good pivot elements for the buckets depends on the elements produced by a particular binary search tree bulk operation. This means that a combiner cannot put elements into buckets before sampling the output collection, to choose pivot elements for the buckets.

```
trait ConcurrentMap[K, V] {
  def put(key: K, value: V): Unit
}

class ConcurrentMapCombiner[K, V](underlying: ConcurrentMap[K, V])
extends Combiner[(K, V), ConcurrentMap[K, V], ConcurrentMapCombiner[K, V]] {
  def +=(elem: (K, V)) {
    underlying.put(elem._1, elem._2)
  }
  def combine(that: ConcurrentMapCombiner[K, V]) = this
  def result = underlying
}
```

Figure 2.14: Concurrent Map Combiner Implementation

For these reasons, a binary search tree combiner works in three steps:

- *Sampling step.* Each combiner appends (using `+=`) elements to a catenable data structure (e.g. Conc-trees described in Chapter 3), called *output elements*, and separately samples some number s of randomly chosen elements, called *sample elements*. Both the output elements and samples are concatenated when `combine` is called. Importantly, note that the output elements are not sorted – they are concatenated in an arbitrary order at this point.
- *Bucket step.* Upon calling the `result` method, the sample elements are used to select some number t of pivot elements, where t is up to an order of magnitude larger than P. The outputs are independently, in parallel, divided into t buckets by each of the P processors. The buckets are merged together in the same way as with hash tables and hash tries.
- *Tree step.* The resulting t buckets can now be used to independently, in parallel, construct t binary search trees, with non-overlapping element ranges. Trees from separate processors can thus be merged in $O(P \log n)$ time, merging each pair of trees as described previously.

2.4.3 Concurrent Combiners

Most existing data-parallel frameworks use concurrent data structures to implement data-parallel transformer operations. In this approach the intermediate data structure is a concurrent data structure shared by all the workers. The intermediate data structure is directly returned when calling the `result` method. This approach typically results in lower performance since every `+=` operation incurs synchronization costs, but is acceptable when the useful workload (e.g. computation in the function passed to the `map` operation) exceeds synchronization costs.

As shown in Figure 2.14, concurrent combiners delegate most of the work to the underlying concurrent data structure. All combiners created in a specific parallel operation instance are instantiated with the same `underlying` data structure, in this case a `ConcurrentMap`. We show a concrete implementation of a concurrent map in Section 4.2.2.

2.5 Data Parallel Bulk Operations

The standard Scala collections framework offers wide range of collection bulk operations [Odersky(2009)]. The goal of a data-parallel collections framework is to implement versions of these operations running in parallel.

We divide these operations into groups that outline the important implementation aspects. For each group we show how to implement them using splitters and combiners described in the previous section. We note that these groups are not disjoint – some operations may belong to several groups.

Trivially Parallelizable Operations

The simplest data-parallel operation is the `foreach` method.

```
def foreach[U](f: T => U): Unit
```

The `foreach` operation takes a higher-order function `f` and invokes that function on each element. The return value of `f` is ignored. The `foreach` method has two properties. First, there are no dependencies between workers working on different collection subsets. Second, it returns no value. Because of these properties, `foreach` is trivially parallelizable – workers do not need to communicate while processing the elements or at the end of the computation to merge their results.

When `foreach` is invoked, a new *task* object is created and submitted to the Fork/Join pool. To split the elements of the collection into subsets, the framework invokes the `split` method of its splitter. Two new child tasks are created and each is assigned one of the child splitters. These tasks are then asynchronously executed by potentially different workers – we say that the tasks are forked. The splitting and forking of new child tasks continues until splitter sizes reach a threshold size. At that point splitters are used to traverse the elements – function `f` is invoked on elements of each splitter. Once `f` is invoked on all the elements, the corresponding task ends. Another example of a method that does not return a value is `copyToArray`, which copies the elements of the of the collection into a target array.

Scalar Result Operations

Most data-parallel operations return a resulting value. The `reduce` operation shown earlier applies a binary associative operator to elements of the collection to obtain a reduction of all the values in the collection:

```
def reduce[U >: T](op: (U, U) => U): U
```

The `reduce` operations takes a binary operator `op`. If, for example, the elements of the collection are numbers, `reduce` can take a function that adds its arguments. Another example is concatenation for collections that hold strings or ropes. Operator `op` must be associative, because the grouping of subsets of elements is undeterministic. Relative order of the elements can be preserved by the data-parallel scheduler, so this operator does not have to be commutative.

The `reduce` operation is implemented in the same manner as `foreach`, but once a task ends, it must return its result to the parent task. Once all the children of the parent task complete, the `op` operator is used to merge the results of the children tasks. Other methods implemented in a similar manner are `aggregate`, `fold`, `count`, `max`, `min`, `sum` and `product`.

Communicating Operations

In the previously shown operations different collection subsets are processed independently. Here we show operations where results computed in one of the tasks can influence the computation of the other tasks. One example is the `forall` method:

```
def forall(p: T => Boolean): Boolean
```

This method only returns `true` if the predicate argument `p` returns `true` for all elements. Sequential collections take advantage of this fact by ceasing to traverse the elements once an element for which `p` does not hold is found. Parallel collections have to communicate that the computation may stop. The parallel `forall` operations must share an invocation-specific context with a flag that denotes whether the computation may stop. When the `forall` encounters an element for which the predicate is not satisfied, it sets the flag. Other tasks periodically check the flag and stop processing elements if it is set.

Operations such as `exists`, `find`, `startsWith`, `endsWith`, `sameElements` and `corresponds` use the same mechanism to detect if the computation can end before processing all the elements. Merging the results of these tasks usually amounts to a logical operation.

Another operation we examine here is `prefixLength`:

```
def prefixLength(p: T => Boolean): Int
```

This operation takes a predicate and returns the length of the longest collection prefix such that all its elements satisfy the predicate `p`. Once some worker finds an element e that does not satisfy the predicate, not all tasks can stop. Workers that operate on

```
159 def setIndexFlagIfLesser(f: Int) = {
160   var loop = true
161   do {
162     val old = READ(flag)
163     if (f >= old) loop = false
164     else if (CAS(flag, old, f)) loop = false
165   } while (loop)
166 }
```

Figure 2.15: Atomic and Monotonic Updates for the Index Flag

parts of the sequence preceding *e* may still find prefix length to be shorter, while workers operating on the following subsequences cannot influence the result and may terminate. To share information about the element's exact position, the invocation-specific context contains an integer flag that can be atomically set by different processors. In Figure 2.15 we show the `setIndexFlagIfLesser` method, which is used by different processors to maintain the index of the leftmost element not satisfying the predicate.

The `setIndexFlagIfLesser` method is a simple example of a concurrent protocol with several interesting properties. First of all, it is linearizable. The linearization point is the only write performed by the method in line 164. Then, the `setIndexFlagIfLesser` method is lock-free. If the `CAS` operation in line 164 fails, then we know that the value of `flag` has changed since the last read. It follows that in the finite number of steps between the last read in line 162 and the `CAS` in line 164 some other thread completed the operation. Finally, the updates by this method to `flag` are monotonic. Because of the check in line 163, the value of the flag can only be decreased and there is no risk of the ABA problem [Herlihy and Shavit(2008)].

Other methods that use integer flags to relay information include `takeWhile`, `dropWhile`, `span`, `segmentLength`, `indexWhere` and `lastIndexWhere`.

Transformer Operations

Certain operations have collections as result types. The standard Scala collection framework calls such operations *transformers*. A typical transformer operation is `filter`:

```
def filter(p: T => Boolean): Repr
```

The `filter` operation returns a collection containing only those elements for which the predicate `p` is true. This operation uses combiners to merge filtered subsets coming from different processors. Methods such as `map`, `take`, `drop`, `slice`, `splitAt`, `zip` and `scan` have the additional property that the resulting collection size is known in advance. This information can be used in specific collection classes to override default implementations and benefit from increased performance. Other methods cannot predict the size of the resulting collection include `flatMap`, `collect`, `partition`, `takeWhile`, `dropWhile`, `span` and `groupBy`.

Multiple-Input Operations

Some data-parallel operations need traverse multiple collections simultaneously. Method `psplit` in `PreciseSplitter`s for parallel sequences is more general than `split`, as it allows splitting a parallel sequence splitter into subsequences of specific lengths. Some methods such as `zip` rely on this capability:

```
def zip[S](that: ParSeq[S]): ParSeq[(T, S)]
```

Operation `zip` returns a sequence composed of corresponding pairs of elements of `this` sequence and another sequence `that`. The regular `split` method would make implementation of this method quite difficult, since it only guarantees to split elements into subsets of arbitrary size – `that` may be a parallel sequence whose splitter produces subsets of different sizes than subsets in `this`. In this case it would not be not clear which elements are the corresponding elements for the pairs. The refined `psplit` method allows both sequences to be split into subsequences of the same sizes. Other methods that rely on precise splitting are `startsWith`, `endsWith`, `patch`, `sameElements` and `corresponds`.

Multiple-Phase Operations

Certain data-parallel operations cannot be completed in a single phase. The `groupBy` operation takes a user-specified operation `f` that assigns each element to some key of type `K`. It returns a parallel map that maps each key value `k` into a sequence of elements `x` of the original collection for which `f(x) == k`:

```
def groupBy[K](f: T => K): ParMap[K, Seq[T]]
```

This operation needs to proceed in two steps. In the first phase each of the workers produces a `HashMapCombiner` shown earlier, such that the elements are grouped into a certain number of buckets according to their keys. Note that the number of buckets is smaller than the number of potential value of type `K`. These `HashMapCombiner`s are merged into a single `HashMapCombiner`. In the second phase workers are assigned to specific buckets and can independently populate regions of a hash map similar to the `result` method of the `HashMapCombiner`.

Another example of a multiple phase operation is `scan`.

2.6 Parallel Collection Hierarchy

In this section we discuss how to organize the collection hierarchy for data-parallel collection frameworks and how to integrate them with existing sequential collection modules in an object-oriented language. We will study two different approaches to doing this – in the first approach new parallel collection classes are introduced for each corresponding sequential collection class, and in the second parallel operations are piggy-backed to existing sequential collection through the use of extension methods. We will refer to the former as *tight integration* [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky], and to the latter as *loose integration* [Prokopec and Petrashko(2013)]. Both approaches add data-parallel operations without changing the existing sequential collection classes, but retain the same signatures for all the bulk operation methods.

We assume that sequential collections have bulk operations that guarantee sequential access. This means that calling a bulk operation such as `foreach` on a collection guarantees that the body of the foreach will be executing on the same thread on which `foreach` was called and that it will be executed completely for one element before processing the next element. Parallel collections have variants of the same bulk operations, but they do not guarantee sequential access – the `foreach` may be invoked simultaneously on different elements by different threads.

Note that sequential access is not related to ordering semantics of the operations. Data-parallel operations can still ensure ordering given that the splitters produce two substrings[3] of the elements in the original collection and that combiners merge intermediate results produced by different workers in the same order as they were split.

Tight Integration

Referential transparency is the necessary condition for allowing a parallel collection to be a subtype of a sequential collection and preserving correctness for all programs. Since Scala and most other general purpose programming languages are not referentially transparent and allow side-effects, it follows that the program using a sequential collection may produce different results than the same program using a parallel collection. If parallel collection types are subtypes of sequential collections, then this violates the Liskov substitution principle, as clients that have a reference statically typed as a sequential collection can get different results when that reference points to a parallel collection at runtime.

To be able to have a reference to a collection which may be either sequential or parallel, there has to exist a common supertype of both collection types. The Scala standard

[3] Note that substrings in the mathematical sense mean that both the order and the contiguity of the original sequence is preserved. This is unlike subsequences in which only the relative order of the elements is preserved.

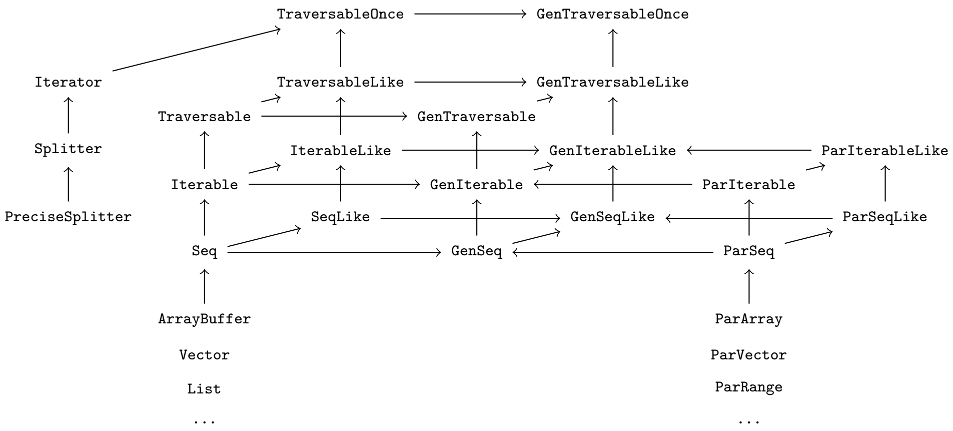


Figure 2.16: Scala Collections Hierarchy

library collection hierarchy defines the most general collection type `GenTraversableOnce` that describes types that can be traversed only once. It is the supertype of both sequential collections and iterators. Collection types that can be traversed arbitrarily many times are represented with the trait `GenTraversable`, and collections that can produce iterators are subtypes of `GenIterable`. Maps, sets and sequences are represented with types `GenMap`, `GenSet` and `GenSeq`, respectively. Sequential collection types that guarantee sequential access, as well as parallel collection types that do not, are subtypes of the corresponding `Gen*` traits. The hierarchy is shown in Figure 2.16, with maps and sets trait omitted for clarity. For example, a `ParSeq` and `Seq` are both subtypes of a general sequence `GenSeq`, but they are in no direct subtyping relationship with respect to each other. The `*Like` types exist to carry the covariant concrete collection type parameter and guarantee the most specific return type principle for collection operations [Odersky and Moors(2009)].

Clients can refer to sequential sequences using the `Seq` trait and to parallel sequences using the `ParSeq` trait. To refer to a sequence whose implementation may be either parallel or sequential, clients can use the `GenSeq` trait. Note that introducing this hierarchy into an existing collection framework preserves source compatibility with existing code – the meaning of all existing programs remains the same.

General collection traits introduce methods `seq` and `par` that return the corresponding sequential or parallel version of the collection, respectively.

```
trait GenIterable[T] {         trait Iterable[T]                trait ParIterable[T]
  def seq: Iterable[T]         extends GenIterable[T] {         extends GenIterable[T] {
  def par: ParIterable[T]        def seq: Iterable[T] = this      def seq: Iterable[T]
}                                def par: ParIterable[T]          def par: ParIterable[T] = this
                               }                                }
```

Each parallel collection class is usually a wrapper around the corresponding sequential collection implementation. Parallel collections require `Combiner`s which extend the `Builder`s used by sequential collections. They therefore implement a method `newCombiner`, and have `newBuilder` defined on sequential collections simply forward the call to `newCombiner`. Similarly, parallel collections define a `splitter` method that subsumes the `iterator` method found on sequential collections.

This design has several disadvantages. First, it clutters the collection hierarchy with additional classes, making the API harder to understand. Second, it requires every sequential collection to be convertible into a parallel collection by calling `par`. As we saw earlier, data structures that satisfy certain parallelism constraints can be converted with thin wrappers, but any non-parallelizable data structure must be converted into a parallelizable data structure by sequentially traversing it when `par` is called, and this cost is not apparent to the user. Parallel collections are required to implement methods such as `reduceLeft` whose semantics do not allow a parallel implementation, leading to confusion. Finally, this design ties the signatures of parallel collections and sequential collections tightly together as they are shared in the `Gen*` traits. If we want to add additional *implicit parameters* to parallel operations, such as data-parallel schedulers, we have to change the signatures of existing sequential collection classes.

Loose Integration

Another approach to augmenting existing parallel collections classes is through extension methods. Here, collection classes exist only for the sequential version of the collection. All the parallel operations are added to this sequential collection class through an implicit conversion.

One of the goals is that the data-parallel operations have the same names and similar signatures as the corresponding sequential operations. This prevents us from adding extension methods directly to sequential collections classes, as it would result in name clashes. Instead, users need to call the `par` method to obtain a parallel version of the collection, as in tight integration. In loose integration, this call returns a thin wrapper of type `Par[Repr]` around the collection of type `Repr`. The `par` method itself is added through an implicit conversion and can be called on any type.

The `Par[Repr]` type has only a few methods — the only interesting one is `seq` that converts it back to the normal collection. Implicit conversions add data-parallel operations to `Par[C]` only for specific collections that are parallelizable.

```
class Par[Repr](val seq: Repr)

implicit class ParOps[Repr](r: Repr) {
  def par = new Par(r)
}
```

```
implicit class ParArrayOps[T](a: Par[Array[T]]) {
  def reduce[U >: T](op: (U, U) => U): U = ???
  def foreach[U](f: T => U): Unit = ???
}
```

This design offers several advantages with respect to tight integration – the signature no longer has to exactly correspond to the sequential counterparts of the parallel methods (e.g. parallel operations now take an additional implicit `Scheduler` parameter), and additional operations that are not part of this collections framework can easily be added by clients to different collections. Similarly, not all sequential collections are required to have their parallel counterparts and inherently sequential bulk operations no longer need to be a part of the parallel interface.

A disadvantage is that clients cannot use parallel collections generically this way. With sequential collections a user can write a function that takes a `Seq[Float]` and pass subtypes like `Vector[Float]` or `List[Float]` to this method. This is possible since `List[Float]` is a subtype of `Seq[Float]`. In the new design this is no longer possible, because a `Par[Vector[Float]]` is not a subtype of `Par[Seq[Float]]`, and for a good reason – it is not known whether parallel operations can be invoked on the `Par[Seq[Float]]` type.

```
def mean(a: Seq[Float]): Float = {
  val sum = a.reduce(_ + _)
  sum / a.length
}
mean(Vector(0.0f, 1.0f, 2.0f))
mean(List(1.0f, 2.0f, 4.0f))
```

```
def mean(a: Par[Seq[Float]]): Float = {
  val sum = a.reduce(_ + _)
  sum / a.length
}
mean(Vector(0.0f, 1.0f, 2.0f).par)
mean(List(1.0f, 2.0f, 4.0f).par) // error
```

To allow code generic in the collection type, loose integration approach defines two special types called `Reducible[T]` and `Zippable[T]`. These traits allow writing generic collection code, since they are equipped with extension methods for standard collection operations, just like `ParArrayOps` are in the example above. The `Reducible[T]` type represents all collections that support parallel operations implementable in terms of a parallel `aggregate` method. The `Zippable[T]` trait is a subtype of `Reducible[T]` and represents that support parallel operations implementable in terms of the parallel `zip` method.

2.7 Task Work-Stealing Data-Parallel Scheduling

Workload scheduling is essential when executing data-parallel operations on multiple processors. Scheduling is the process of assigning parts of the computational workload to different processors. In the context of this thesis, the parts of the computational workload are separate collection elements. Data-parallel scheduling can be done offline, before the data-parallel operation starts, or online, during the execution of a data-parallel operation. This thesis focuses on runtime data-parallel scheduling.

Data-parallel operations are performed on collection elements so scheduling can be done by partitioning the collection into element subsets. Implementing a parallel `foreach` method from Section 2.2 requires that subsets of elements are assigned to different processors. These subsets can be assigned to different threads – each time a user invokes the `foreach` method on some collection, a thread is created and assigned a subset of elements to work on. However, thread creation is expensive and can exceed the cost of the collection operation by several orders of magnitude. For this reason it makes sense to use a pool of worker threads in sleeping state and avoid thread creation each time a parallel operation is invoked.

There exists a number of frameworks that implement thread pools, a prominent one being the Java Fork/Join Framework [Lea(2000)] that is now a part of JDK. Fork/Join Framework introduces an abstraction called a Fork/Join task which describes a unit of work to be done. This framework also manages a pool of worker threads, each being assigned a queue of Fork/Join tasks. Each task may spawn new tasks (`fork`) and later wait for them to finish (`join`).

The simplest way to schedule work between processors is to divide it into fixed-size chunks and schedule an equal part of these on each processor. There are several problems with this approach. First of all, if one chooses a small number of chunks, this can result in poor workload-balancing. Assuming that some of the elements have a lot more work associated with them than the others, a processor may remain with a relatively large chunk at the end of the computation, and all other processors may have to wait for it to finish. Alternatively, a large number of chunks guarantees better granularity and load-balancing, but imposes a higher overhead, since each chunk requires some scheduling resources. One can derive expressions for theoretically optimal sizes of these chunks [Kruskal and Weiss(1985)], but the driving assumptions for those expressions assume a large number of processors, do not take scheduling costs into account and ignore effects like false-sharing present in modern multiprocessor systems. Other approaches include techniques such as *guided self scheduling* [Polychronopoulos and Kuck(1987)] or *factoring* [Hummel et al.(1992)Hummel, Schonberg, and Flynn].

An optimal execution schedule may depend not only on the number of processors and data size, but also on irregularities in the data and processor availability. Since these circumstances cannot be anticipated in advance, runtime information must be used to guide load-balancing. Task-based work-stealing [Blumofe and Leiserson(1999)] has been a method of choice for many applications that require runtime load-balancing.

In task-based work-stealing, work is divided to tasks and distributed among workers (typically processors or threads). Each worker maintains a task queue. Once a processor completes a task, it dequeues the next one. If its queue is empty, the worker tries to steal a task from another worker's queue. This topic has been researched in depth, and in the context of this thesis we rely on the Java Fork/Join Framework to schedule asynchronous computation tasks [Lea(2000)].

Lets assume that the amount of work per element is the same for all elements. We call such a data-parallel workload *uniform*. Making tasks equally sized guarantees that the longest idle time is equal to the time to process one task. This happens if all the processors complete when there is one more task remaining. If the number of processors is P, the work time for $P = 1$ is T and the number of tasks is N, then equation 2.1 denotes the theoretical speedup in the worst case.

$$speedup = \frac{T}{(T - T/N)/P + T/N} \underset{P \to \infty}{\to} N \tag{2.1}$$

Thread wake-up times, synchronization costs, memory access costs and other real-world effects have been omitted from this idealized analysis. In practice, there is a considerable overhead with each created task – we cannot expect to have a work-stealing task for each element. Fewer tasks reduce the overheads of scheduling, but reduce the potential for parallelization.

More importantly, some parts of a parallel computation always need to execute sequentially, and comprise a serial bottleneck of the parallel computation. For example, creating the top-level task object and waking up the worker threads cannot be parallelized. The serial bottlenecks impose an upper bound on the possible speedup from parallelization. This is known as the *Amdahl's Law*, and is captured in the following equation, where $T(x)$ is the execution time of the parallel program when it is executed by x processors, B is the fraction of the program that has to execute serially, and P is the number of processors in the system:

$$speedup = \frac{T(1)}{T(P)} = \frac{1}{B + \frac{1}{P} \cdot (1 - B)} \underset{P \to \infty}{\to} \frac{1}{B} \tag{2.2}$$

This simple observation tells us that, regardless of the number of processors P in the system, we can never accelerate the program by more than $\frac{1}{B}$ times.

Data-parallel workloads are in practice irregular (i.e. there is a different amount of work associated with each element), not only due to the properties of the data in the underlying program, but also due to different processor speeds, worker wakeup time, memory access patterns, managed runtime mechanisms like garbage collection and JIT compilation, interaction with the underlying operating system and other causes. In practice every data-parallel operation executed on multiple CPUs has an irregular workload to some degree.

We found that *exponential task splitting* is effective when load-balancing data-parallel operations [Cong et al.(2008)Cong, Kodali, Krishnamoorthy, Lea, Saraswat, and Wen]. The main idea behind this technique is the following. If a worker thread completes its work with more tasks in its queue that means other workers have not been stealing tasks from it. The reason why other workers are not stealing tasks is because they are preoccupied

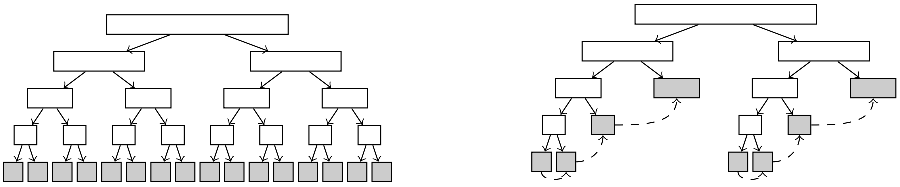


Figure 2.17: Fine-grained and exponential task splitting

with work of their own, so it is safe to assume that they will continue to be busy in the future – the worker does more work with the next task. The heuristic is to double the amount of work before checking again (Figure 2.17). However, if the worker thread does not have more tasks in its queue, then it must steal a task from another worker. The stolen task is always the biggest task on a queue.

There are two points worth mentioning here. First, stealing tasks is generally more expensive than popping them from the worker's own queue. Second, the Fork/Join framework allows only the oldest tasks on the queue to be stolen. The former means the less times stealing occurs, the better – we will want to steal bigger tasks. The latter means that which task gets stolen depends on the order tasks were pushed to the queue (forked). We will thus push the largest tasks first. Importantly, after a task is stolen, it gets split until reaching some threshold size. This allows other workers to potentially steal tasks. This process is illustrated in Figure 2.17.

We show the pseudocode for *exponential task splitting* in Figure 2.18, where an abstract Fork/Join task called `Batch[R]` is shown. Its abstract methods are implemented in specific data-parallel operations.

Before we discuss the pseudocode, we note that two additional optimizations have been applied in Figure 2.18. When splitting a task into two tasks, only one of them is pushed to the queue, and the other is used to recursively split as part of computation. Pushing and popping to the queue involves synchronization, so directly working on a task improves performance. Furthermore, Java Fork/Join Framework support unforking tasks previously put to the task queue by calling `tryUnfork`. After a worker finishes with one of the tasks, it tries to unfork the previously forked task in line 20 and work on it directly instead of calling `join`.

Once a data-parallel operation is invoked on a collection, a Fork/Join task executes its `compute` method in line 9. The `compute` method may decide that this task needs to be divided into smaller tasks, and calls the method `internalTask` to do this. The `internalTask` method in turn calls `unfold` to divide the collection into smaller parts. The corresponding collection is then split into two parts by the `split` method. The `right` child task is chained to the list of forked tasks in line 28 using its `next` pointer. The `right` task is then forked in line 29. Forking a task means that the task gets pushed

```
1  abstract class Batch[R] extends RecursiveTask[R] {
2    var next: Batch[R] = null
3    var result: R
4    def split: (Batch[R], Batch[R])
5    def merge(that: Batch[R]): Unit
6    def mustSplit: Boolean
7    def leafTask(): Unit
8    def compute() {
9      if (mustSplit) internalTask()
10     else leafTask()
11   }
12   def internalTask() = {
13     var last = unfold(this, null)
14     last.leafTask()
15     result = last.result
16     fold(last.next)
17   }
18   def fold(last: Batch[T]) {
19     if (last != null) {
20       if (last.tryUnfork()) last.leafTask()
21       else last.join()
22       this.merge(last)
23       fold(last.next)
24     }
25   }
26   def unfold(head: Batch[R], last: Batch[R]): Batch[R] = {
27     val (left, right) = head.split
28     right.next = last
29     right.fork()
30     if (head.mustSplit) unfold(left, right)
31     else {
32       left.next = right
33       left
34     }
35   }
36 }
```

Figure 2.18: Exponential Task Splitting Pseudocode

on the processor's task queue. The `left` child task is split recursively until a threshold governed by the abstract `mustSplit` method is reached – at that point subset of elements in the smallest `left` task is processed sequentially in the `leafTask` call in line 7, which corresponds to the leftmost leaf of the computation tree in Figure 2.17. This call also assigns the result of processing that particular batch of elements to its `result` field. After finishing with one task, the worker tries to unfork a task from its queue if that is possible by calling `tryUnfork` in line 20. In the event that unforking is not possible due to the task being stolen and worked on by another processor, the worker calls `join`. Once the `last` task is known to have its `result` field set, the worker merges its result with the result in the current task by calling the abstract `merge` method.

Since tasks are pushed to the queue, the last (smallest) task pushed will be the first task popped. At any time the processor tries to pop a task, it will be assigned an amount of work equal to the total work done since it started with the leaf. On the other hand, if there is a processor without tasks in its own queue, it will steal from the opposite side of the queue. When a processor steals a task, it again divides that task until it reaches the threshold size.

The worst case scenario is that a worker pops the last remaining, biggest task from its queue. We know this task came from the processor's own queue (otherwise it would have been split, enabling the other processors to steal and not be idle). At this point the processor will continue working for some time T_L. If we assume that the input data is relatively uniform, then T_L must be approximately equal to the time spent up to that moment. This is because the tasks are split in two recursively, so the biggest task has approximately the same number of elements as all the smaller tasks. If the task size is fine-grained enough to be divided among P processors, work up to that moment took $(T - T_L)/P$, so $T_L = T/(P+1)$. Total time for P processors is then $T_P = 2T_L$. The equation 2.3 gives a bound on the worst case speedup, assuming $P \ll N$, where N is the number of generated tasks:

$$\mathit{speedup} = \frac{T}{T_P} = \frac{P+1}{2} \tag{2.3}$$

This estimate says that if the workload is relatively uniform, then the execution time is never more than twice as great as the lower limit, given that the biggest number of tasks generated is $N \gg P$. To ensure that there are sufficient tasks, we define the minimum task size as $\mathit{threshold} = \max(1, \frac{n}{8P})$, where n is the number of elements to process.

An important thing to notice here is that the threshold controls the maximum number of tasks that get created. Even if the biggest tasks from each task queue always get stolen, the execution degenerates to the balanced computation tree shown in figure 2.17. The likelihood of this to happen has shown to be small in practice and exponential splitting generates less tasks than dividing the collection into equal parts.

The task-based work-stealing data-parallel scheduling shown in this section does effective load-balancing for relatively uniform data-parallel workloads. In Chapter 5 we show a more complex work-stealing scheduling technique specifically designed for irregular data-parallel computations.

2.8 Compile-Time Optimisations

The genericity of the data-parallel collection framework shown so far comes with abstraction costs that are sometimes unacceptable. In this section we classify such costs and describe ways to deal with them.

The most important data-parallel collection framework requirement is that it allows the clients to write faster programs. Programs can then be made faster by running them on more processors – we say that a data-parallel framework needs to be *scalable*. However, even if it scales well, there is little use of a parallel collection framework that needs several processors to be as fast as the optimal sequential collections framework. Today's desktop processors have a dozen or less cores, so it is wasteful to use them to overcome inefficiencies in the framework.

Another important reason to optimize data-parallel operations is to better understand the scalability of a parallel algorithm. An optimal data-parallel program often has very different performance characteristics than a non-optimal one. In a non-optimal data-parallel program, computational resources are spent parallelizing executions which are not present in an optimal data-parallel program, often resulting in different scalability characteristics.

Abstraction penalties are one example of an execution present in a non-optimal program. To illustrate how non-optimal executions, such as abstraction penalties, affect scalability, we show two versions of a simple data-parallel `reduce` operation. We run both of them on the Intel 3.40 GHz Quad Core i7-2600 processor. In the first version, we call `reduce` on a hash table containing integers. The first version uses the classic Scala Parallel Collections framework [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky], which incurs boxing and other indirection penalties. In the second version, we call `reduce` on an array containing integers. The second version uses the ScalaBlitz framework [Prokopec et al.(2014b)Prokopec, Petrashko, and Odersky], described later in this section. ScalaBlitz relies on Scala Macros to eliminate abstraction penalties related to boxing and using iterators. The ScalaBlitz framework uses the `toPar` method to denote a parallel execution.

```
hashset.par.reduce(_ + _)                 array.toPar.reduce(_ + _)
```

The `hashset` contains $5M$ elements in the Parallel Collections version, and the `array` contains $50M$ elements in the ScalaBlitz version. The graph in Figure 2.19 shows two curves, which reflect the running times of the two `reduce` operations. Note that we chose different collection sizes to bring the two curves closer together. The Parallel Collection version scales nicely – the speedup for four processors is 2.34×. The ScalaBlitz framework scales much worse for this workload – the speedup is only 1.54×. From this, we might conclude that Parallel Collections framework is better at data-parallel scheduling, but this is not the case. The reason why ScalaBlitz is unable to obtain a better speedup is not because processors are idling without work, but because they are unable to simultaneously retrieve more data from the main memory. The Intel 3.40 GHz Quad Core i7-2600 processor has a dual-channel memory controller – when the processors spend most of their time fetching data from memory and almost no time working on the data (as is the case with the `_ + _` operator for the `reduce`), the maximum parallelism level that this architecture can support is 2. By contrast, the Parallel Collections version spends time boxing and unboxing data elements, checking the (open-addressing) hash table for empty entries, and updating the local iterator state. The memory bus traffic is,

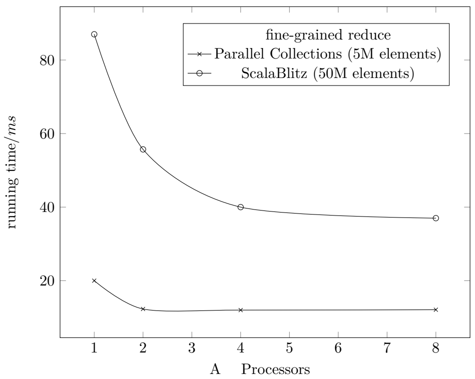


Figure 2.19: ScalaBlitz and Parallel Collections for Fine-Grained Uniform Workloads

in this case, amortized by useless operations induced by abstraction penalties, hence, we observe better scalability regardless of the worse absolute performance.

This performance effect is not a random artifact of the ScalaBlitz framework. We can manually reproduce it by starting P threads, each of which traverses and sums P contiguous disjoint subintervals, with size $\frac{n}{P}$, of an array with size n. On the Intel 3.40 GHz Quad Core i7-2600 processor, we will not obtain a speedup larger than 2. In comparison, the UltraSPARC T2 processor, with four dual-channel memory controllers, exhibits a much nicer scalability in this example.

We feel that this is a canonical example of how a suboptimal framework exhibits different performance characteristics than an optimized one. Upon optimizing the program, a separate set of performance effects becomes evident, and impacts the program's performance characteristics, such as scalability, in unforeseen ways.

This example should be a convincing argument why optimising data-parallel operations is important before starting to reason about their scalability. A data-parallel operation that is 10× slower than the optimal one can be perfectly scalable, but this does not mean that the same data-parallel operation is faster than the corresponding optimal sequential version. Before making any claims about the scalability of a data-parallel framework, we should always make sure that the framework is optimal when compared to the sequential baseline.

2.8.1 Classifying the Abstraction Penalties

The task of implementing an efficient data-parallel framework is made hard by the fact that data-parallel frameworks offer genericity on several levels. First, parallel operations are generic both in the type of the data records and the way these records are processed. Orthogonally, records are organized into data sets in different ways depending on how they are accessed – as arrays, hash-tables or trees. Let us consider the example of a subroutine that computes the mean of a set of measurements to illustrate these concepts.

We show both its imperative and data-parallel variant.

```
def mean(x: Array[Int]) = {
  var sum = 0
  while (i < x.length) {
    sum += x(i); i += 1
  }
  return sum / x.length
}
```

```
def mean(x: Array[Int]) = {
  val sum = x.par.fold(0) {
    (acc, v) => acc + v
  }
  return sum / x.length
}
```

The data-parallel operation that the declarative-style `mean` subroutine relies on is `fold`, which aggregates multiple values into a single value. This operation is parametrized by the user-specified aggregation operator. The data set is an array and the data records are the array elements, in this case integers. A naive implementation of a parallel `fold` method might be as follows:

```
37 def fold[T](xs: ParIterable[T], z: T, op: (T, T) => T) = {
38   val (left, right) = xs.splitter.split
39   def par(s: Splitter[T]) = forkAndJoin {
40     var sum = z
41     while (s.hasNext) sum = op(sum, s.next())
42     sum
43   }
44   return op(par(left), par(right))
45 }
```

This `fold` implementation assumes we have a uniform workload and only 2 processors, so it divides the `splitter` of `xs` into two child splitters. These `left` and `right` child splitters are processed in parallel – from a high-level perspective, this is done by the `forkAndJoin` call. Once all the workers complete, their `results` can be aggregated sequentially. We focus on the work done by separate workers, namely, lines 40 through 42. Note that the `while` loop in those lines resembles the imperative variant of the method `mean`, with several differences. The neutral element of the aggregation `z` is generic and specified as an argument. Then, instead of comparing a local variable `i` against the array length, method `hasNext` is called, which translates to a dynamic dispatch. The second dynamic dispatch updates the state of the splitter and returns the `next` element and another dynamic dispatch is required to apply the summation operator to the integer values.

These inefficiencies are referred to as the *abstraction penalties*. We can identify several abstraction penalties in the previous example. First of all, in typical object-oriented languages such as Java or C++ the dynamic dispatches amount to reading the address of the virtual method table and then the address of the appropriate method from that table. Second, and not immediately apparent, the splitter abstraction inherently relies on maintaining the traversal continuation. The method `next` must read an integer field, check the bounds and write the new value back to memory before returning the corresponding value in the array. The imperative implementation of `mean` merely reads the array value and updates `i` in the register. The third overhead has to do with representing method parameters in a generic way. In languages like Java, Scala and OCaml primitive values passed to generic methods are converted to heap objects and their references are used instead. This is known as *boxing* and can severely impact peformance. While in languages like C++ templating can specialize the `fold` for primitive types, generic type parameters are a problem for most managed runtimes.

It is important to note that most of the computational time in a data-parallel operation is spent in the lines 40 through 42, which we refer to as the *batch processing loop*. Eliminating the abstraction penalties in the batch processing loop is the key to achieving efficient parallelization.

2.8.2 Operation and Data Structure Type Specialization

In the context of the JVM compilation techniques were proposed to eliminate boxing selectively, like the *generic type specialization* transformation used in Scala [Dragos(2010)] [Dragos and Odersky(2009)]. Most of the abstractions introduced so far were parametrized by the type of the elements contained in the collections. For example, `Splitter`s have a type parameter `T` denoting the type of the elements contained in corresponding collections. Abstractions and internals of a data-parallel framework that interact with those abstractions must be specialized with the `@specialized` annotation [Prokopec et al.(2014b)Prokopec, Petrashko, and Odersky].

However, generic type specialization focuses exclusively on problems related to erasure. While it can be used to eliminate boxing, it does not eliminate other abstraction penalties.

2.8.3 Operation Kernels

To eliminate unneeded method call indirections and the use of splitters in the batch processing loop, every invocation of a data-parallel operations needs to be specialized for a particular callsite. This specialization involves inlining the splitter logic and the operation parameters into the generic operation body, as was done in the ScalaBlitz framework [Prokopec et al.(2014b)Prokopec, Petrashko, and Odersky].

At every callsite we create a kernel object that optimizes the data-parallel operation. The `Kernel` object describes how a batch of elements is processed and what the resulting value is, how to combine values computed by different workers and what the neutral element for the result is. The kernel interface is shown in Figure 2.20. The method `apply` takes the splitter as the argument. It uses the splitter to traverse its elements and compute the result of type `R`. The method `combine` describes how to merge two different results and `zero` returns the neutral element.
How these methods work is best shown through an example of a concrete data-parallel operation. The `foreach` operation takes a user-specified function object `f` and applies it

```
trait Kernel[T, R] {
  def zero: R
  def combine(a: R, b: R): R
  def apply(it: Splitter[T]): R
}
```

Figure 2.20: The `Kernel` Interface

in parallel to every element of the collection. Assume we have a collection `xs` of integers and we want to assert that each integer is positive:

```
xs.foreach(x => assert(x > 0))
```

The generic `foreach` implementation is as follows:

```
def foreach[U](f: T => U): Unit = {
  val k = new Kernel[T, Unit] {
    def zero = {}
    def combine(a: Unit, b: Unit): Unit = {}
    def apply(it: Splitter[T]) =
      while (it.hasNext) f(it.next())
  }
  invokeParallel(k)
}
```

The `Unit` type indicates no return value – the `foreach` function is executed merely for its side-effect, in this case a potential assertion. Methods `zero` and `combine` always return the `Unit` value `()` for this reason. Most of the processing time is spent in the `apply` method, so its efficiency drives the running time of the operation. We use the Scala Macro system [Burmako and Odersky(2012)] to inline the body of the function `f` into the `Kernel` at the callsite:

```
def apply(it: Splitter[T]) =
  while (it.hasNext) assert(it.next() > 0)
```

Another example is the `fold` operation mentioned in the introduction and computing the sum of a sequence of numbers `xs`:

```
xs.fold(0)((acc, x) => acc + x)
```

Operation `fold` computes a resulting value, which has the integer type in this case. Results computed by different workers have to be added together using `combine` before returning the final result. After inlining the code for the neutral element and the body of the folding operator, we obtain the following kernel:

```
new Kernel[Int, Int] {
  def zero = 0
  def combine(a: Int, b: Int) = a + b
  def apply(it: Splitter[Int]) = {
    var sum = 0
    while (it.hasNext) sum = sum + it.next()
    sum
  }
}
```

While the inlining shown in the previous examples avoids a dynamic dispatch on the function object, the `while` loop still contains two virtual calls to the splitter. Maintaining the splitter requires writes to memory instead of registers. It also prevents optimizations like loop-invariant code motion, e.g. hoisting the array bounds check necessary when the

```
def apply(it: RangeSplitter) =
  var sum = 0
  var p = it.i
  val u = it.until
  while (p < u)
    sum = sum + p
    p += 1
  return sum
```

```
def apply(it: ArraySplitter[T]) =
  var sum = 0
  var p = it.i
  val u = it.until
  while (p < u)
    sum = sum + array(p)
    p += 1
  return sum
```

Figure 2.21: The Specialized `apply` of the `Range` and `Array` Kernels for `fold`

```
def apply(it: TreeSplitter[T]) = {
  def traverse(t: Tree): Int =
    if (t.isLeaf) t.element
    else traverse(t.left) + traverse(t.right)
  val root = i.nextStack(0)
  traverse(it.root)
}
```

Figure 2.22: The Specialized `apply` of the `Tree` Kernel for `fold`

splitter traverses an array.

For these reasons, we would like to inline the iteration into the `apply` method itself. This, however, requires knowing the specifics of the data layout in the underlying data-structure. Within this section we rely on the macro system to apply these transformations at compile-time – we will require that the collection type is known statically to eliminate the `next` and `hasNext` calls.

`IndexKernel`. Data-structures with fast indexing such as arrays and ranges can be traversed efficiently by using a local variable `p` as iteration index. Figure 2.21 shows range and array kernel implementations for the `fold` example discussed earlier. Array bounds checks inside a `while` loop are visible to the compiler or a runtime like the JVM and can be hoisted out. On platforms like the JVM potential boxing of primitive objects resulting from typical functional object abstractions is eliminated. Finally, the dynamic dispatch is eliminated from the loop.

`TreeKernel`. The tree splitters introduced in Section 2.3 assumed that any subtree can be traversed with the `next` and `hasNext` calls by using a private stack. Pushing and popping on this private stack can be avoided by traversing the subtree directly. Figure 2.22 shows a kernel in which the `root` of the subtree is traversed with a nested recursive method `traverse`.

`HashKernel`. The hash-table kernel is based on an efficient `while` loop like the array and range kernels, but must account for empty array entries. Assuming flat hash-tables with linear collision resolution, the `while` loop in the kernel implementation of the previously mentioned `fold` is as follows:

```
while (p < u) {
  val elem = array(p)
  if (elem ≠ null) sum = sum + elem
  p += 1
}
```

Splitter implementations for hash-tables based on closed addressing are similar.

2.8.4 Operation Fusion

Basic declarative operations such as `map`, `reduce` and `flatMap` allow expressing a wide range of programs, often by pipelining these combinators together. The downside of pipelining is that it allocates intermediate collections that can be eliminated through loop fusion. In this section we give several examples on how operations can be fused together.

A common pattern seen in programs is a transformer operation like a `map` followed by an aggregation like `reduce`:

```
def stdev(xs: Array[Float], mean: Float): Float =
  xs.map(x => sqr(abs(x - mean))).reduce(_ + _)
```

A pattern such as a `map` call followed by a `reduce` call can be detected in code and replaced with a more efficient `mapReduce` operation as follows:

```
def stdev(xs: Array[Float], mean: Float): Float =
  xs.mapReduce(x => sqr(abs(x - mean)))(_ + _)
```

Another example are *for*-comprehensions in Scala that allow expressing data queries more intuitively. Assume we have a list of employee incomes and tax rates, and we want to obtain all the tax payments for the employees. The following *for*-comprehension does this:

```
for (i <- income.toPar; t <- taxes) yield i * t
```

The Scala compiler translates this into `map` and `flatMap` calls:

```
income.toPar.flatMap(i => taxes.map(t => i * t))
```

We want to avoid having intermediate collections for each employee generated by the `map`. To do this, before generating the kernel `apply` method, we rewrite the kernel code according to the following translation scheme:

```
xs.map(f).foreach(g) ⇒ inline[xs.foreach(x => g(f(x)))]
```

Above we rewrite any two subsequent applications of a `map` with a mapping function `f` and a `foreach` with a traversal function `g` to a single `foreach` call with a traversal function obtained by fusing and inlining `f` and `g`. Thus, the `while` part of the array transformer kernel `apply` following from `flatmap` above:

```
while (p < u) {
  taxes.map(t => array(p) * t).foreach(s => cmb += s)
  p += 1
}
```

that uses a combiner `cmb`, is rewritten to:

```
while (p < u) {
  taxes.foreach(t => cmb += array(p) * t)
  p += 1
}
```

An additional rewrite rule for `flatMap` and `foreach` expressions allows us to transform arbitrarily nested *for*-comprehensions into nested `foreach` loops.

```
xs.flatMap(f).foreach(g) ⇒ inline[xs.foreach(x => f(x).foreach(g))]
```

While the rewrite-rule approach does not cover many optimizations opportunities that a more complex dataflow analysis can address, it optimizes many bottlenecks and offers significant benefits in performance.

2.9 Linear Data Structure Parallelization

In the previous sections we focused on parallelizable data structures, i.e. data structures that can be divided into subsets in $O(\log n)$ time or better, where n is the number of elements in the data structure. Not all common data structures fit into this category. Indeed, data structures such as linked lists are unsuitable for parallelization methods seen so far, as splitting an arbitrary linked list requires traversing half of its elements. For many operation instances such an $O(n)$ splitting is unacceptable – in this case it might be more efficient to first convert the linked list into an array and then use splitters on the array.

In this section we study alternative approaches to parallelizing collection operations on linear data structures like linked lists and lazy streams. We note that the parallelization for these data structures does not retain information about the relative order of elements – the operators applied to their data-parallel operations need to be commutative.

2.9.1 Linked Lists and Lazy Streams

A linked list data structure consists of a set of nodes such that each node points to another node in the set. A special node in this set is considered to be the `head` of the linked list. We consider connected linked lists without loops – every node can be reached from the `head`, no two nodes point to the same node and no node points at the root. Lazy streams are similar to linked lists with the difference that some suffix of the lazy stream might not be computed yet – it is created the first time it is accessed.

```
 74 class Invocation[T](xs: List[T]) {
 75   @volatile var stack = xs
 76   def READ = unsafe.getObjectVolatile(this, OFFSET)
 77   def CAS(ov: List[T], nv: List[T]) = unsafe.compareAndSwapObject(this, OFFSET, ov, nv)
 78 }
 79 
 80 abstract class ListTask[T] extends RecursiveTask[T] {
 81   val inv: Invocation[T]
 82   def workOn(elem: T): Unit
 83   def compute() {
 84     val stack = inv.READ
 85     stack match {
 86       case Nil =>
 87         // no more work
 88       case head :: tail =>
 89         if (inv.CAS(stack, tail)) workOn(stack)
 90         compute()
 91     }
 92   }
 93 }
 94 
 95 class ListForeach[T, U](f: T => U, val inv: Invocation[T]) extends ListTask[T] {
 96   def workOn(elem: T) = f(elem)
 97 }
 98 
 99 implicit class ListOps[T](val par: Par[List[T]]) {
100   def foreach[U](f: T => U) = {
101     val inv = new Invocation(par.seq)
102     val tasks = for (i <- 0 until P) yield new ListForeach[T](f, inv)
103     for (t <- tasks) t.fork()
104     for (t <- tasks) t.join()
105   }
106 }
```

Figure 2.23: Parallel List Operation Scheduling

We show a way to parallelize linked list operations in Figure 2.23, where the linked list itself is used as a stack of elements. In the concrete example above we use the immutable `List`s from the Scala standard library. The stack is manipulated using atomic READ and CAS operations [4] to ensure that every element is assigned to exactly one worker. After reading the `stack` field in line 84, every worker checks whether it is empty, indicating there is no more work. If the list is non-empty, every worker attempts to replace the current list by its `tail` with the CAS line 89. Success means that only that worker replaced, so he gains ownership of that element – he must call `workOn` to process the element. If the CAS is not successful, the worker retries by calling the tail-recursive `compute` again.

The algorithm is correct – a successful CAS in line 89 guarantees that the `stack` field has not changed its value since the READ in line 84. No linked list suffix will ever appear more than once in the field `stack` because there are no more loops in the linked list when following the `tail` pointer. Furthermore, every linked list node will appear in the `stack` field at least once since the list is connected. Together, these properties ensure

[4] The atomic READ and the CAS operation are implemented in terms of `sun.misc.Unsafe` object that allows executing low-level JVM operations on memory addresses computed with the `this` pointer and the `OFFSET` of a particular object field. In the later sections we avoid this implementation detail and implicitly assume that operations like READ and CAS exist on a particular data structure.

that every linked list element is assigned to exactly one worker.

The algorithm shown in Figure 2.23 is *lock-free* – although many threads compete to update the field `stack` at the same time, after a certain finite number of steps some thread will always complete its operation. In general this number of steps can depend on the size of the data structure or the number of competing workers, but in this case the number of steps is constant and equals the number of instructions between two consecutive CAS calls in line 89.

2.9.2 Unrolled Linked Lists

The scheduling for linked lists shown in the previous section is lock-free, but it has two main disadvantages. First, it is inefficient for low per-element workloads. Second, it is not scalable since workers need to serialize their accesses to the `stack` field. It is thus suitable only for operations where the per-element workload is much higher than the scheduling cost – one should not attempt to compute a scalar product of two vectors this way.

Unrolled linked lists consist of a set of nodes, each of which contains a contiguous array of elements, called a *chunk*. In essence, the type `Unrolled[T]` is equivalent to `List[Array[T]]`. Unrolled linked list amortize the scheduling costs of algorithm in Figure 2.23 by processing all the elements in the chunk in the `workOn` call. For low per-element workloads this makes the scheduler efficient after a certain chunk size, but not scalable – as the number of workers rise hardware cache-coherency protocols become more expensive, raising the minimum chunk size for efficient scheduling. A similar contention effect is shown later on a concrete experiment in Figure 5.7 of Chapter 5, where the number of elements in the chunk is shown on the x-axis and the time required to process the loop on the y-axis.

2.10 Related Work

General purpose programming languages and the accompanying platforms currently provide various forms of library support for parallel programming. Here we give a short overview of the related work in the area of data parallel frameworks, which is by no means comprehensive.

Data-parallelism is a well-established concept in parallel programming languages dating back to APL in 1962 [Iverson(1962)], subsequently adopted by NESL [Blelloch(1992)], High Performance Fortran and ZPL. With the emergence of commodity parallel hardware data parallelism is gaining more traction. Frameworks like OpenCL and CUDA focusing mainly on GPUs are heavily oriented towards data parallelism, however, they impose certain programming constraints such as having to avoid general recursion and nested data-parallelism. Chapel [Chamberlain(2013)] is a parallel programming language supporting both task and data parallelism that improves the separation between data-structure implementation and algorithm description. Many other frameworks and languages adopt data parallelism.

.NET langugages have support for common parallel programming patterns, such as parallel looping constructs, aggregations and the map/reduce pattern [Toub(2010)]. These constructs relieve the programmer of having to reimplement low-level details such as correct load-balancing between processors each time a parallel application is written. The .NET Parallel LINQ framework provides parallelized implementations of .NET query operators. On the JVM, the Java `ParallelArray` [Lea(2014)] is an early example of a data-parallel collections. JDK 8 Parallel Streams are a recent addition, which parallelizes bulk operations over arbitrary data structures. Data Parallel Haskell has a parallel array implementation with parallel bulk operations [Peyton Jones(2008)]. Some frameworks have so far recognized the need to employ divide and concquer principles in data structure design. Fortress introduces conc-lists, a tree-like list representation which allows expressing parallel algorithms on lists [Steele(2010)]. In this chapter, we generalized their traversal concept to maps and sets, and both mutable and immutable data structures.

Intel TBB [Reinders(2007)] for C++ bases parallel traversal on iterators with splitting and uses concurrent containers to implement transformer operations. Operations on concurrent containers are slower than their sequential counterpartss. STAPL for C++ has a similar approach – they provide thread-safe concurrent objects and iterators that can be split [Buss et al.(2010)Buss, Harshvardhan, Papadopoulos, Amato, and Rauchwerger]. The STAPL project also implements distributed containers. Data structure construction is achieved by concurrent insertion, which requires synchronization.

X10 [Charles et al.(2005)Charles, Grothoff, Saraswat, von Praun, and Sarkar] comes with both JVM and C backends providing task and data parallelism, while Fortress targets the JVM, supports implicit parallelism and a highly declarative programming style. JVM-based languages like Java and Scala now provide data-parallel support as part of the standard library.

2.11 Conclusion

In this chapter, we provided parallel implementations for a wide range of operations found in the Scala standard collection library. We did so by introducing two simple divide and conquer abstractions called *splitters* and *combiners*. These abstractions were sufficient to implement most bulk operations on most collection types. In specific cases, such as the `map` operation on a parallel array, more optimal operations can be implemented by special-casing the operation implementation. In other cases, such as linear data structures like linked lists, splitters could not be efficiently implemented. Different parallelization schemes are possible in such circumstances, which may incur higher synchronization costs, thus making the sequential baseline unachievable. Regardless of these exceptions, we learned that the splitter/combiner model serves as a good foundation for data-parallelism in most situations.

Having learned about the basics of implementing a data-parallel collection framework, we turn to more specific topics in the following chapters. Relying on what we learned so far, we will explore data structures that are more suitable for data-parallelism, investigate parallelization in the absence of quiescence and bulk-synchronous operation mode, and see how to schedule data-parallel workloads that are particularly irregular. Throughout the rest of the thesis, we apply the fundamentals presented in this chapter, and use them as a guiding principle when designing efficient data structures and algorithms for data-parallelism.

3 Conc-Trees

In this chapter we investigate several data structures that are better suited for data-parallelism. Note that balanced trees are of particular interest here. They can be efficiently split between CPUs, so that their subsets are independently processed, and they allow reaching every element in logarithmic time. Still, providing an efficient tree concatenation operation and retaining these properties is often challenging. Despite the challenge, concatenation is essential for implementing declarative data-parallel operations, as we have learned when we introduced combiners in Chapter 2.

Concatenation is also required when parallelizing functional programs, which is one of the design goals in Fortress. In the following we compare a *cons-list*-based functional implemenation of the `sum` method against the *conc-list*-based parallel implementation [Steele(2009)]:

```
def sum(xs: List[Int]) =
  xs match {
    case head :: tail =>
      head + sum(tail)
    case Nil => 0
  }
```

```
def sum(xs: Conc[Int]) =
  xs match {
    case ls <> rs =>
      sum(ls) + sum(rs)
    case Single(x) => x
  }
```

The first `sum` implementation decomposes the data structure `xs` into the first element `head` and the remaining elements `tail`. The sum is computed by adding `head` to the sum of the `tail`, computed recursively. While efficient, this implementation cannot be efficiently parallelized. The second `sum` implementation decomposes `xs` into two parts `ls` and `rs`, using the (for now hypothetical) `<>` extractor. It then recursively computes partial sums of both parts before adding them together. Assuming that `xs` is a balanced tree, the second `sum` implementation can be efficiently parallelized.

Perfectly balanced trees have optimal depth and minimize traversal time to each element, but are problematic when it comes to updating them. For example, appending an element to a *complete binary tree*, and maintaining the existing ordering of elements is an $O(n)$ operation. Ironically, this expensive property does not substantially increase the performance of lookup operations, so more relaxed balancing guarantees are preferred in practice.

In this chapter, we describe the a relaxed binary tree data-structure called *Conc-tree*, used for storing sequences of elements, and several variants of the Conc-tree data structure. The basic variant of the data structure is *persistent* [Okasaki(1998)], but we use it to design efficient mutable data structures. More concretely:

- We describe a Conc-tree lists with *worst-case* $O(\log n)$ time *persistent* insert, remove and lookup operations, and *worst-case* $O(\log n)$ *persistent* split and concatenation operations.
- We describe the Conc-tree rope variant with optimal traversal and memory usage, and introduce *amortized* $O(1)$ time *ephemeral* append and prepend operations.
- We describe the conqueue data structure with *amortized* $O(1)$ time *ephemeral* deque operations. We then describe the lazy conqueue data structure with *worst-case* $O(1)$ time *persistent* deque operations. Both data structures retain the *worst-case* $O(\log n)$ bound for splitting and concatenation.
- We show how to implement mutable buffers and deques using various Conc-tree variants, reducing their constant factors to a minimum.

In Section 3.1, we introduce Conc-tree lists. We discuss Conc-tree ropes in Section 3.2. In Section 3.3, we study conqueues and lazy conqueues. We conclude the chapter by applying Conc-trees to several combiners introduced in Chapter 2. These modifications will improve existing combiner implementations by a constant factor and yield a significant speedup in data-parallel transformer operations.

3.1 Conc-Tree Lists

As noted in Chapter 2, balanced trees are good for parallelism. They can be efficiently split between processors and allow reaching every element in logarithmic time. Providing an efficient merge operation and retaining these properties is often challenging. In this section we focus on the basic Conc-tree data structure that stores sequences of elements and provides a simple, fast $O(\log n)$ concatenation.

Perfectly balanced trees have optimal depth and minimize traversal time to each element, but they are problematic when it comes to updating them. For example, appending a single element to a *complete binary tree*[1] (while maintaining the existing ordering of

[1] In a complete binary tree all the levels except possibly the last are completely full, and nodes in the last level are as far to the left as possible.

```
abstract class Conc[+T] {
  def level: Int
  def size: Int
  def left: Conc[T]
  def right: Conc[T]
  def normalized = this
}

abstract class Leaf[T]
extends Conc[T] {
  def left = error()
  def right = error()
}


case object Empty extends Leaf[Nothing] {
  def level = 0
  def size = 0
}

case class Single[T](x: T) extends Leaf[T] {
  def level = 0
  def size = 1
}

case class <>[T](left: Conc[T], right: Conc[T])
extends Conc[T] {
  val level = 1 + left.level.max(right.level)
  val size = left.size + right.size
}
```

Figure 3.1: Basic Conc-Tree Data Types

elements) is an $O(n)$ operation. Ironically, this expensive property does not substantially increase the performance of lookup operations.

Trees with relaxed invariants are typically more efficient to maintain in terms of asymptotic running time. Although they provide less guarantees on their balance, the impact of being slightly imbalanced is small in practice – most trees break the perfect balance by at most a constant factor. As we will see, the Conc-tree list will have a classic relaxed invariant seen in red-black and AVL trees – the longest path from the root to a leaf is never more than twice as long as the shortest path from the root to a leaf.

The Conc-tree data structure may be composed of several types of nodes. We will denote that node type of the Conc-tree as `Conc`. This abstract data type will have several concrete data types, similar to how the functional `List` data type is either an empty list `Nil` or a `::` (pronounced *cons*) of an element and another list. The `Conc` may either be an `Empty`, denoting an empty tree, a `Single`, denoting a tree with a single element, or a `<>` (pronounced *conc*), denoting two separate subtrees.

We show these basic data types in Figure 3.1. Any `Conc` has an associated `level`, which denotes the longest path from the root to some leaf in that tree. The `level` is defined to be 0 for the `Empty` and `Single` tree, and 1 plus the level of the deeper subtree for the `<>` tree. The `size` of a `Conc` denotes the total number of elements contained in the Conc-tree. The `size` and `level` are cached as fields in the `<>` type to prevent traversing the tree to compute them each time they are requested. `Conc` trees are immutable like cons-lists – they are never modified after construction. We defer the explanation of the `normalized` method until Section 3.2 – for now `normalized` just returns the tree.

It is easy to see that the data types described so far can yield trees that are not balanced. First of all, we can construct arbitrarily large empty trees by combining the `Empty` tree instances with `<>`. We will thus enforce the following invariant – the `Empty` tree can never be a part of `<>`, as it is a special case that can only be a Conc-tree on its own. However, this restriction is not sufficient to make trees balanced. Here is another simple example in which construct a Conc-tree by iteratively adding elements to the right:

```
(0 until n).foldLeft(Empty: Conc[Int])((tree, x) => new <>(tree, new Single(x)))
```

What are the sufficient conditions to make this tree balanced? Similar to AVL trees [Adelson-Velsky and Landis(1962)], we will require that the difference in `levels` of the left subtree and the right subtree is never greater than 1. This relaxed invariant imposes the following bounds on the number of elements in the tree with the height *level*. Assume first that the tree is completely balanced, i.e. every `<>` node has two chilren of equal `level`. In this case the size $S(level)$ of a subtree at a particular *level* is:

$$S(level) = 2 \cdot S(level - 1), S(0) = 1 \tag{3.1}$$

This recurrence is easy to solve – $S(level) = 2^{level}$. If we denote the number of elements in the tree as $n = S(level)$, it follows that the level of this tree is $level = \log_2 n$. Now, assume that the tree is in a very relaxed state – every `<>` node at a specific *level* has two subtrees such that $|left.level - right.level| = 1$. The size of a node at *level* is then:

$$S(level) = S(level - 1) + S(level - 2), S(0) = 1 \tag{3.2}$$

This equation should be familiar to most readers, as it is one of the first recurrences most people meet. It is the Fibonacci recurrence with the following solution:

$$S(level) = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{level} - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^{level} \approx \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{level} \tag{3.3}$$

where the second addend becomes insignificant for larger values of *level*, so we ignore it. The level of such a tree with n elements is $level = \log_{\frac{1+\sqrt{5}}{2}} n + \log_{\frac{1+\sqrt{5}}{2}} \sqrt{5}$.

From the monotonicity of the recurrences above it follows that $O(\log n)$ is both an upper and a lower bound for the Conc-tree depth, assuming we keep the difference in sibling heights less than or equal to 1. The upper bound, although logarithmic, is somewhat unfriendly to the possible maximum depth, as the base of this logarithm is approximately 1.618. Experimental data suggests that most Conc-trees are much less skewed than that – in fact, AVL and red-black trees have the same depth bounds. Balanced trees can be efficiently parallelized and, as explained in Section 2.3, Conc-trees are size-combining trees. The bounds also ensure that Conc-trees have $O(\log n)$ indexing and update operations. Their implementation is straightforward:

```
def apply(xs: Conc[T], i: Int) = xs match {
  case Single(x) => x
  case left <> right =>
    if (i < left.size) apply(left, i)
    else apply(right, i - left.size)
}
def update(xs: Conc[T], i: Int, y: T) =
  xs match {
    case Single(x) => Single(y)
    case left <> right if i < left.size =>
      new <>(update(left, i, y), right)
    case left <> right =>
      val ni = i - left.size
      new <>(left, update(right, ni, y))
  }
```

From the way that the indexing operation `apply` is defined it follows that a left-to-right in-order traversal visits the elements of the Conc-tree – a `foreach` operation looks very similar. The `update` operation produces a new Conc-tree such that the element at index `i` is replaced with a new element `y`. This operation only allows replacing existing Conc-tree elements and we would like to able to insert elements into the Conc-tree as well. Before showing an $O(\log n)$ insert operation implementation we will study a way to efficiently concatenate two Conc-trees together.

We define the implicit class `ConcOps` to provide nicer concatenation syntax – the expression `xs <> ys` concatenates two Conc-trees together:

```
implicit class ConcOps[T](val xs: Conc[T]) {
  def <>[T](ys: Conc[T]) = {
    if (xs == Empty) ys else if (ys == Empty) xs
    else concat(xs.normalized, ys.normalized)
  }
}
```

The expression `xs <> ys` is different than the expression `new <>(xs, ys)`, which would simply link the two trees together with one `<>` node – invoking the `new` constructor directly can violate the balance invariant. We thus refer to composing two trees together with a `<>` node as *linking*. Creating a Conc-tree that respects the invariants and represents the concatenated sequence of the two input trees we term *concatenation*.

The bulk of the concatenation logic is in the `concat` method, which is shown in Figure 3.2. This method assumes that the two trees are *normalized*, i.e. composed from the basic data types in Figure 3.1 and respecting the invariants.

In explaining the code in Figure 3.2 we will make an assumption that concatenating two Conc-trees can yield a tree whose `level` is either equal to the larger input Conc-tree or greater by exactly 1. In other words, concatenation never increases the Conc-tree `level` by more than 1. We call this the *height-increase assumption*. We will inductively show that the height-increase assumption is correct while explaining the recursive `concat` method in Figure 3.2. We skip the trivial base case of merging `Single` trees.

```
64 def concat[T](xs: Conc[T], ys: Conc[T]) = {
65   val diff = ys.level - xs.level
66   if (abs(diff) <= 1) new <>(xs, ys)
67   else if (diff < -1) {
68     if (xs.left.level >= xs.right.level) {
69       val nr = concat(xs.right, ys)
70       new <>(xs.left, nr)
71     } else {
72       val nrr = concat(xs.right.right, ys)
73       if (nrr.level == xs.level - 3) {
74         val nr = new <>(xs.right.left, nrr)
75         new <>(xs.left, nr)
76       } else {
77         val nl = new <>(xs.left, xs.right.left)
78         new <>(nl, nrr)
79       }
80     }
81   } else {
82     if (ys.right.level >= ys.left.level) {
83       val nl = concat(xs, ys.left)
84       new <>(nl, ys.right)
85     } else {
86       val nll = concat(xs, ys.left.left)
87       if (nll.level == ys.level - 3) {
88         val nl = new <>(nll, ys.left.right)
89         new <>(nl, ys.right)
90       } else {
91         val nr = new <>(ys.left.right, ys.right)
92         new <>(nll, nr)
93       }
94     }
95   }
96 }
```

Figure 3.2: Conc-Tree Concatenation Operation

The trees `xs` and `ys` may be in several different relationships with respect to their `level`s. We compute their difference in `level`s into a local value `diff`, and use it to disambiguate between three cases. First of all, the absolute difference between the `level`s of `xs` and `ys` could differ by one or less. This is an ideal case – the two trees can be linked directly by creating a `<>` node that connects them. In this case, concatenating two trees is a constant time operation.

Otherwise, one tree has a greater `level` than the other one. Without the loss of generality we assume that the left Conc-tree `xs` is higher than the right Conc-tree `ys`. To concatenate `xs` and `ys` we need to break `xs` into parts, and concatenate these parts in a different order before linking them into a larger tree. Depending on whether `xs` is left-leaning or right-leaning, we proceed by cases.

First, let's assume that `xs.left.level >= xs.right.level`, that is, `xs` is left-leaning. The concatenation `xs.right <> ys` in line 69 does not increase the height of the right subtree by more than 1. This means that the difference in `level`s between `xs.left` and `xs.right <> ys` is 1 or less, so we can link them directly in line 70. We prove this by the following sequence of relationships:

$$\begin{aligned}
xs.left_{level} - xs.right_{level} &\in \{0, 1\} && \text{initial assumption} \\
xs_{level} &> ys_{level} + 1 && \text{initial assumption} \\
\Rightarrow xs.right_{level} &\geq ys_{level} && \text{from the balance invariant} \\
\Rightarrow nr_{level} - xs.right_{level} &\leq 1 && \text{from the height-assumption} \\
\Rightarrow nr_{level} - xs.left_{level} &\in \{-1, 0, 1\} && \text{from the initial assumption} \\
\Rightarrow |(xs.left \diamond nr)_{level} - xs_{level}| &\leq 1 && \text{from } xs_{level} - xs.left_{level} = 1
\end{aligned}$$

Thus, under the height-increase assumption the final concatenated tree will not increase its height by more than 1, so we inductively proved that the assumption holds for this case.

Now, let's assume that `xs.left.level < xs.right.level`. We can no longer concatenate the subtrees as before – doing so might result in the balance violation. The subtree `xs.right.right` is recursively concatenated with ys in line 72. Its level may be equal to either `xs.level - 2` or `xs.level - 3`. After concatenation we thus obtain a new tree `nr` with the level anywhere between `xs.level - 3` and `xs.level - 1`. However, if the `nr.level` is equal to `xs.level - 3`, then `xs.right.left.level` is `xs.level - 2` – this follows directly from the balance invariant. Depending on the level of `nr` we either link it with `xs.right.left` or we link `xs.left` with `xs.right.left` before linking the result to `nr`. In both cases the balance invariant is retained.

$$\begin{aligned}
xs.right_{level} - xs.left_{level} &= 1 && \text{initial assumption} \\
xs_{level} &> ys_{level} + 1 && \text{initial assumption} \\
\Rightarrow xs_{level} - xs.right.right_{level} &\in \{2, 3\} && \text{from the balance invariant} \\
\Rightarrow xs.right.right_{level} + 1 &\geq ys_{level} && \text{from the initial assumption} \\
\Rightarrow xs_{level} - nrr_{level} &\in \{1, 2, 3\} && \text{from the height-assumption} \\
xs_{level} - nrr_{level} &= 3 && \\
\Rightarrow xs.right.left.level &= xs_{level} - 2 && \text{from the balance invariant} \\
\Rightarrow (xs.right.left \diamond nrr)_{level} &= xs_{level} - 1 && \\
\Rightarrow (xs.left \diamond (xs.right.left \diamond nrr))_{level} &= xs_{level} && \text{from } xs.left_{level} = xs_{level} - 1 \\
xs_{level} - nrr_{level} &\in \{1, 2\} && \\
\Rightarrow xs_{level} - xs.right.left_{level} &\in \{2, 3\} && \text{from the balance invariant} \\
\Rightarrow (xs.left \diamond xs.right.left)_{level} &= xs_{level} - 1 && \\
\Rightarrow ((xs.left \diamond xs.right.left) \diamond nrr)_{level} &= xs_{level} &&
\end{aligned}$$

Again, due to the height-increase assumption in the subtrees, the resulting concatenated tree does not increase its height by more than 1. This turns the height-increase assumption into the following theorem.

Theorem 3.1 (Height Increase) *Concatenating two Conc-tree lists of heights h_1 and h_2 yields a tree with height h such that $|h - max(h_1, h_2)| \leq 1$.*

Theorem 3.2 (Concatenation Running Time) *Concatenating two Conc-tree lists of heights h_1 and h_2 is an $O(|h_1 - h_2|)$ asymptotic running time operation.*

Proof. Direct linking in the concatenation operation is always an $O(1)$ operation. Recursively invoking `concat` occurs exactly once on any control path in `concat`. Each time `concat` is called recursively, the height of the higher Conc-tree is decreased by 1, 2 or 3. Method `concat` will not be called recursively if the absolute difference in Conc-tree heights is less than or equal to 1. Thus, `concat` can only be called at most $|xs_{level} - ys_{level}|$ times. □

Corollary 3.3 *Concatenating two Conc-trees of heights h_1 and h_2, respectively, allocates $O(|h_1 - h_2|)$ nodes.*

These theorems will have significant implications in proving the running times of data structures shown later in this chapter. We now come back to the `insert` operation to show the importance of concatenation on a simple example. The concatenation operation makes expressing the `insert` operation straightforward:

```
def insert[T](xs: Conc[T], i: Int, y: T) =
  xs match {
    case Single(x) =>
      if (i == 0) new <>(Single(y), xs)
      else new <>(xs, Single(y))
    case left <> right if i < left.size =>
      insert(left, i, y) <> right
    case left <> right =>
      left <> insert(right, i - left.size, y)
  }
```

Insert unzips the tree along a certain path by dividing it into two subtrees and inserting the element into one of the subtrees. That subtree will increase its height by at most one by Theorem 3.1, making the height difference with its sibling at most two. Merging the two new siblings is thus $O(1)$ by Theorem 3.2. Since the length of the path from the root to any leaf is $O(\log n)$, the total amount of work done becomes $O(\log n)$. Note that the complexity of the `insert` method remains $O(\log n)$ specifically because we concatenate the new trees in that order. Had we been linking the trees going top-down instead of bottom-up, the complexity would increase to $O(\log^2 n)$, as we would be concatenating consecutively smaller trees to a large tree that is slowly increasing its depth.

Having understood how `insert` works and why it is an $O(\log n)$ operation, implementing `split` becomes second-nature to us. The major difference is that we have to produce two trees instead of just one:

```
def split[T](xs: Conc[T], n: Int): (Conc[T], Conc[T]) = xs.normalized match {
  case left <> right =>
    if (n < left.size) {
      val (ll, lr) = split(left, n)
      (ll, lr <> right)
    } else if (n > left.size) {
      val (rl, rr) = split(right, n - left.size)
      (left <> rl, rr)
    } else (left, right)
  case s: Single[T] =>
    if (n == 0) (Empty, s)
    else (s, Empty)
  case Empty =>
    (Empty, Empty)
}
```

This implementation is concise, but trips on one unfortunate aspect of the JVM – it is not possible to return both trees at once without allocating a tuple, as the JVM does not yet provide non-boxed value types. In our real implementation, we avoid the need to create tuples by storing one of the resulting trees into a reference cell of type `ObjectRef[Conc[T]]`, which is passed around as a third argument to the `split` method.

Coming back to the benefits of concatenation, prepending and appending elements to a Conc-tree list amounts to merging a `Single` tree with the existing Conc-tree:

```
def <>[T](x: T, xs: Conc[T]) =
  Single(x) <> xs
def <>[T](xs: Conc[T], x: T) =
  xs <> Single(x)
```

The downside of prepending and appending elements like this is that it takes $O(\log n)$ time. While this is generally regarded as efficient, it is not satisfactory if most of the computation involves appending or prepending elements, as is the case with `Combiners` from the last section. We see how to improve this bound next.

3.2 Conc-Tree Ropes

In this section we show a modification of the Conc-tree data structure that supports an amortized $O(1)$ time append operation when used ephemerally. The reason that `append` shown in the last section took $O(\log n)$ time is that it had to traverse the Conc-tree from the root to some leaf. We note that the append position is always the same – the rightmost leaf. However, even if we could expose that rightmost position by defining the Conc-tree as a tuple of the root and the rightmost leaf, updating the path from that leaf to the root would still take $O(\log n)$ time – this is because the Conc-tree is immutable. Instead, we will relax the invariants on the Conc-tree data structure.

First, we introduce a new type of a Conc-tree node called `Append`. The `Append` node shown below has a structure isomorphic to the `<>` node. The difference is that the `Append` node is not required to respect the balance invariant, that is, the heights of its `left` and `right` Conc-trees are not constrained in any way. However, we impose the *append invariant* on `Append` nodes, which says that the `right` subtree of an `Append` node is never another `Append` node. Furthermore, the `Append` tree cannot contain `Empty` nodes. Finally, only an `Append` node may point to another `Append` node. The `Append` node is thus isomorphic to a linked list with the difference that the last node is not `Nil`, but another Conc-tree.

This data type is transparent to the client and could have alternatively been encoded as a special bit in `<>` nodes – clients never observe nor can construct `Append` nodes. A custom extractor ensures that an `Append` node behaves like a `<>` node in a pattern match.

```
case class Append[T](left: Conc[T], right: Conc[T])
extends Conc[T] {
  val level = 1 + left.level.max(right.level)
  val size = left.size + right.size
  override def normalized = wrap(left, right)
}

def wrap[T](xs: Conc[T], ys: Conc[T]) =
  xs match {
    case Append(ws, zs) => wrap(ws, zs <> ys)
    case xs => xs <> ys
  }
```

The `normalized` method behaves differently for `Append` nodes. We define `normalized` to return the Conc-tree that contains the same sequence of elements as the original Conc-tree, but is composed only of the basic Conc-tree data types in Figure 3.1. We call this process *normalization*. The definition in Section 3.1 already does that by returning `this`. The `normalized` method in `Append` calls the recursive `wrap` method. The `wrap` method simply folds the Conc-trees in the linked list induced by `Append`.

We postpone making any claims about the normalization running time for now. Note, however, that the previously defined `concat` method invokes `normalized` twice and is expected to run in $O(\log n)$ time. Thus, the `normalized` method should not be worse than $O(\log n)$ either.

We now return to the append operation, which is supposed to add a single element at the end of the Conc-tree. Recall that by using `concat` directly this operation has $O(\log n)$ running time. We now try to implement a more efficient append operation. The invariant for the `Append` nodes allows appending a new element as follows:

```
def append[T](xs: Conc[T], ys: Single[T]) = new Append(xs, ys)
```

Such an `append` is a constant-time operation, but it has very negative consequences for the `normalized` method. Appending n elements like this results in a long list-like

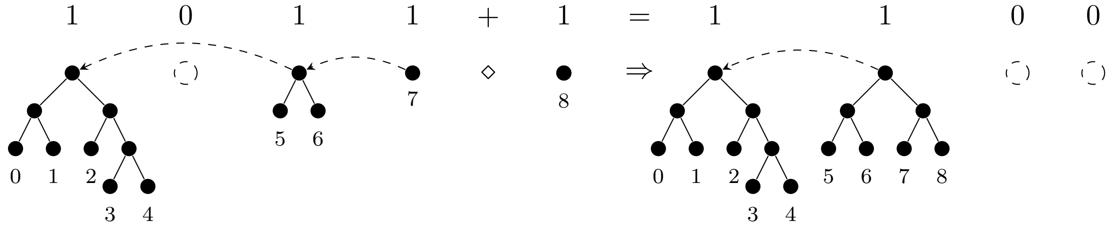


Figure 3.3: Correspondence Between the Binary Number System and Append-Lists

Conc-tree on which the `normalized` method, and hence `concat`, takes $O(n \log n)$ time. This attempt sheds light on the fact that the amount of work done by `append` impacts the complexity of the subsequent concatenation. The more time `append` spends organizing the relaxed Conc-tree, the less time will a `concat` have to spend later when normalizing.

Before attempting a different approach to `append`, we note that there is a correspondence between a linked list of trees of different `level`s and the digits of different weights in a standard binary natural number representation. This correspondence is induced by directly linking two Conc-tree nodes of the same `level` with a new `<>` node, and adding two binary digits of the same weight.

An important property of binary numbers is that counting up to n takes $O(n)$ computational steps, where one computational step is rewriting a single digit in the binary representation. Adding 1 is usually an $O(1)$ operation, but the carries chain-react and occasionally require up to $O(\log n)$ rewrites. It follows that adding n `Single` trees in the same way results in $O(n)$ computational steps, where one computation step is linking two trees with the same `level` together – a constant-time operation.

We therefore expand the append invariant – if an `Append` node `a` has another `Append` node `b` as the left child, then `a.right.level < b.right.level`. If we now interpret the Conc-trees in under `Append` nodes as binary digits with the weight 2^{level} we end up with the sparse binary number representation [Okasaki(1998)]. In this representation zero digits (missing Conc-tree levels) are not a part of the physical structure in memory. This correspondence is illustrated in Figure 3.3, where the binary digits are shown above the corresponding Conc-trees and the dashed line represents the linked list formed by the `Append` nodes.

Figure 3.4 shows the `append` operation that executes in $O(1)$ amortized time. The link operation in line 149, which corresponds to adding to binary digits, occurs only for adjacent trees that happen to have the same `level` – in essence, `append` is the implementation of the carry operation in binary addition. How does this influence the running time of normalization? The trees in the append list are in a form that is friendly to normalization. This list of trees of increasing size is such that the height of the largest tree is $O(\log n)$ and no two trees have the same height. It follows that there are no more than $O(\log n)$ such trees. Furthermore, the sum of the height differences between

```
def append[T](xs: Conc[T], ys: Leaf[T]) =
  xs match {
    case Empty => ys
    case xs: Leaf[T] => new <>(xs, ys)
    case _ <> _ => new Append(xs, ys)
    case xs: Append[T] => append(xs, ys)
  }

private def append[T](xs: Append[T], ys: Conc[T]) =
  if (xs.right.level > ys.level) new Append(xs, ys)
  else {
    val zs = new <>(xs.right, ys)
    xs.left match {
      case ws @ Append(_, _) =>
        append(ws, zs)
      case ws =>
        if (ws.level <= xs.level) ws <> zs
        else new Append(ws, zs)
    }
  }
```

Figure 3.4: Append Operation

adjacent trees is $O(\log n)$. By Theorem 3.1 concatenating any two adjacent trees y and z in the strictly decreasing sequence t^*xyzs^* will yield the tree with height no larger than the height of x. By Theorem 3.2, the total amount of work required to merge $O(\log n)$ such trees is $O(\log n)$. Thus, appending in the same way as incrementing binary numbers yields a list of trees for which normalization runs in the required $O(\log n)$ time.

Note that the public `append` method takes a `Leaf` node instead of a `Single` node. The conc-lists from Section 3.1 and their variant from this section have a high memory footprint. Using a separate leaf to represent each element is extremely inefficient. Not only does it create a pressure on the memory management system, but traversing all the elements in such a data structure is suboptimal. Conc-tree travesal (i.e. a `foreach`) should ideally have the same running time as array traversal, and its memory consumption should correspond to the memory footprint of an array. We therefore introduce a new type of a `Leaf` node in this section called a `Chunk` that packs the elements more tightly together. As we will see in Section 3.4, this will also ensure very efficient mutable `+=` operation.

```
case class Chunk[T](xs: Array[T], size: Int, k: Int) extends Leaf[T] {
  def level = 0
}
```

The `Chunk` node contains an array `xs` containing `size` elements. It has an additional argument `k` that denotes the maximum number of elements that a `Chunk` can have. The `insert` operation from Section 3.1 is modified to copy the target `Chunk` when updating the Conc-tree, and will divide the `Chunk` into two if `size` exceeds `k`. Similarly, a `remove` operation fuses two adjacent `Chunks` if their total size is below a certain threshold.

The *conc-rope* described in this section has one limitation. When used persistently, there is a possibility that we obtain an instance of the Conc-tree whose next `append` triggers a chain of linking operations. If we repetitively use that instance of the tree for appending,

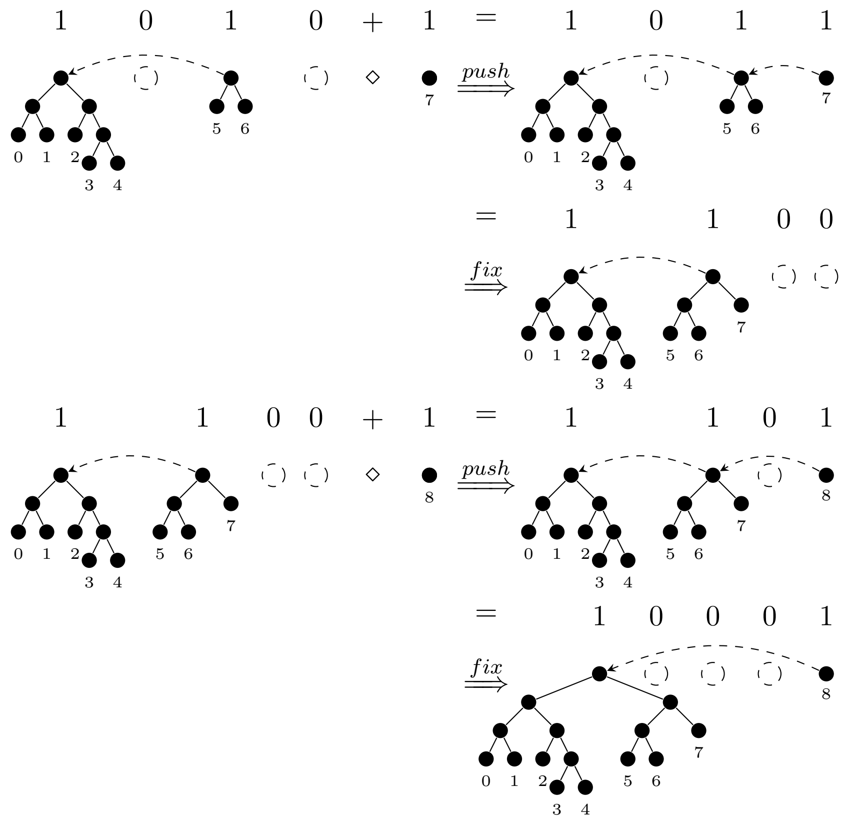


Figure 3.5: Correspondence Between the Fibonacci Number System and Append-Lists

we lose the amortized $O(1)$ running time. Thus, when used persistently, the conc-rope has $O(\log n)$ appends. This data structure deserves to be mentioned nonetheless, since its simplicity ensures good constant factors for its $O(1)$ ephemeral use. In fact, many applications, such as data-parallel combiners, will always use the most recent version of the data structure. For such applications better constant factors from a simpler data structure matter.

Still, it is fair to ask if there is a more efficient variant of Conc-trees that allows $O(1)$ appends and prepends when used persistently, while retaining $O(\log n)$ bounds for concatenation. Removing these limitations is the topic of the next section – in fact, we will show that appends and prepends can execute in $O(1)$ worst-case time.

3.3 Conqueue Trees

In this section we present a variant of the Conc-tree data structure that supports $O(\log n)$ worst-case time persistent concatenations, as well as $O(1)$ worst-case persistent deque operations. Deque (a double-ended queue) allows prepending and appending elements, as well as removing the first or the last element. We will refer to these four operations as pushing and popping the head element or the last element.

Let us recount some of the difficulties with conc-ropes that lead to worst-case $O(\log n)$ operations when conc-ropes were used persistently. The $O(\log n)$ linking steps ensue when appending to a Conc-tree that corresponds to a sequence of 1-digits. In Figure 3.3 this sequence of 1-digits had the length two, but the sequence can have an arbitrary length.

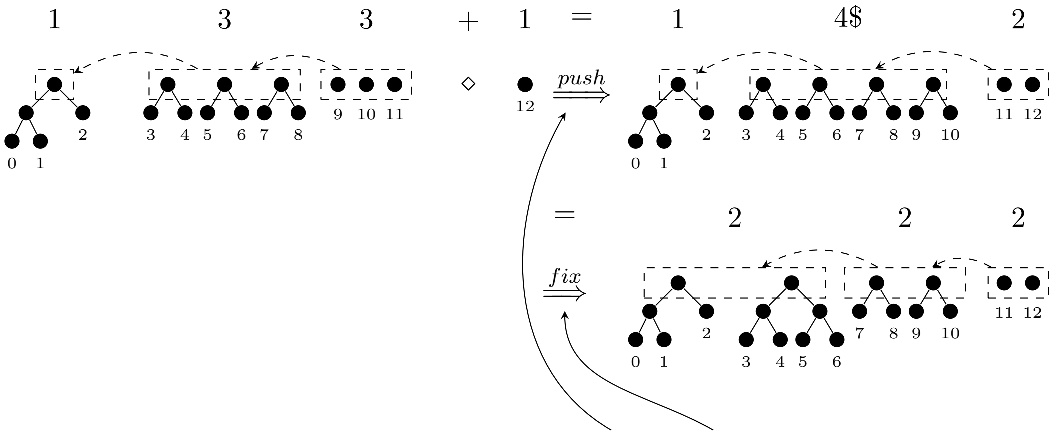
The reason why this happens is that the standard binary number system is too lazy in pushing the carries forward. This number system allows the carry work to accumulate more quickly than it is performed, so, occasionally, incrementing needs a logarithmic amount of time. We need a number system that can push carries more often and avoid arbitrary length 1-sequences – it can serve as a blueprint for designing deque operations.

One such number system is the Fibonacci number system shown in Figure 3.5. In this number system the weight of a digit at position n is the n-th Fibonacci number $F(n)$. Sequence 110 represents the number 5, but so does the sequence 1000 – this number system is redundant. Its important property is that for every number there exists a representation that has no two subsequent 1 digits. We can take advantage of this property by removing all the occurrences of the pattern 11 after every increment. In fact, it is possible to count by removing at most one occurrence of 11 after very increment.

This number system is ideal for Conc-trees, since the transformation between two redundant representations is a simple $O(1)$ linking operation. The balance invariant ensures that we can link two trees adjacent in level. If we remove one occurence of 11 after every append operation, an append operation will never take more than $O(1)$ time. The data structure can keep track of the lowest occurrence of the 11 pattern with an auxiliary stack – at any point there might be several such occurrences.

However, the push operation defined like this merges the Conc-trees in the append-list too often. As shown in that last tree in Figure 3.5, at some point we end up with a long sequence of 0-digits at the right side of the append-list – in this state a pop operation needs $O(\log n)$ time, because a tree with `level` 1 needed for the pop operation is buried $O(\log n)$ pointer hops away in the first non-empty Conc-tree (i.e. the rightmost 1-tree labeled 7 in the last append-list in Figure 3.5). It turns out this counting approach is too eager in pushing the carries forward.

Allowing efficient pop operations requires not only pushing carries as far as possible, but also keeping digits of different weights nicely distributed. We need a number system that is tuned just right – increments and decrements must do enough, but not too much work.



$\ldots \to 23 \to 32 \to 33 \to 122 \to 123 \to 132 \to 133 \to 14\$2 \equiv 222 \to 223 \to 232 \to \ldots$

Figure 3.6: Correspondence Between the 4-Digit Base-2 System and Append-Lists

4-Digit Number System with Base 2

Interestingly, the deque based on the Fibonacci number system described above can be modified to eliminate only occurrences of patterns 111 and 000. Although this approach allows $O(1)$ pop operations, it has the following disadvantage. Removing occurrences of 000 patterns requires tearing apart the tree preceding the pattern to fill in the gap. In a number system with two digits this means that the tree preceding the 000 gap must break in two trees of differing heights to replace the gap with the new pattern 110. This means that every tree has to be left-leaning and this is problematic for deque operations, which would have to break trees in either direction.

Instead of insisting on a number system with two digits, we show a positional number system with four digits in which a digit at position i has weight 2^i. This number system is redundant – the number 5 can be represented both as 13 or as 21. Counting up in this number system starts with numbers 0, 1, 2 and 3. Incrementing 3 requires a carry, so we must combine two digits with weight $2^0 = 1$ into a new 1 with weight $2^1 = 2$. We end up with the number 12. Continuing to count like this gives us numbers 13, 22, 23, 32, 33, 122 and so on.

Figure 3.6 shows counting up in this number system and how it corresponds to Conc-tree lists – the bar at the bottom shows individual computational steps (i.e. rewrites) during counting. The computational steps marked with ≡ denote carries. We use the 4$ notation to indicate the position in the number where a carry must occur. The digit 4 is never actually a part of the number, it merely represents an intermediate state in the rewrites.

One append-list is not sufficient to implement a deque, since it allows push and pop operations only at one end. The deque implementation in this section will keep track of two sequences of Conc-trees each corresponding to a 4-digit number with base 2. These sequences will meet at their ends and occasionally exchange Conc-trees at the point

```
abstract class Num[+T] extends Conc[T] {
  def leftmost: Conc[T]
  def rightmost: Conc[T]
  def rank: Int
}
case object Zero extends Num[Nothing] {
  def leftmost = error()
  def rightmost = error()
  def left = error()
  def right = error()
  def level = 0
  def size = 0
  def rank = 0
  override def normalized = Empty
}
case class Three[T]
  (_1: Conc[T], _2: Conc[T], _3: Conc[T])
extends Num[T] {
  def leftmost = _1
  def rightmost = _3
  def left = _1
  lazy val right = _2 <> _3
  def level = 1 +
    max(_1.level, _2.level, _3.level)
  def size = _1.size + _2.size + _3.size
  def rank = 3
  override def normalized = _1 <> _2 <> _3
}
```

```
abstract class Conqueue[+T]
extends Conc[T] {
  def evaluated: Boolean
}
case class Tip[T](tip: Num[T])
extends Conqueue[T] {
  def left = tip.left
  def right = tip.right
  def level = tip.level
  def size = tip.size
  def evaluated = true
  def rear = error()
  override def normalized =
    tip.normalized
}
case class Spine[+T](
  lwing: Num[T], rwing: Num[T],
  lazy val rear: Conqueue[T]
) extends Conqueue[T] {
  def left = lwing
  lazy val right = new <>(rear, rwing)
  lazy val level = 1 +
    max(lwing.level, rear.level, rwing.level)
  lazy val size =
    lwing.size + rear.size + rwing.size
  def evaluated = isEvaluated(rear)
  override def normalized = wrap(this)
}
```

Figure 3.7: Conqueue Data Types

where they meet, as if the whole data structure is one big conveyer belt. Later, we will show that this deque implementation also supports $O(\log n)$ time concatenation. We will call the deque with efficient concatentation *a conqueue*.

3.3.1 Basic Operations

In this section we show the conqueue push-head and pop-head operations. Note that the push-last and pop-last operations are their mirrored versions. Similar to the conc-rope shown in Section 3.2 we will introduce several new data types for conqueues that rely on the basic Conc-trees from Figure 3.1. We show the conqueue data types in Figure 3.7. Conqueue data types are divided into two groups. The `Num`, or *number*, data types on the left correspond to digits in our number system. We only show implementations for `Zero` and `Three` – `One` and `Two` are similar. The `Num` data type requires the `leftmost` and the `rightmost` method that simply returns the leftmost or the rightmost Conc-tree contained in it. Its `rank` operation returns the integer value of the corresponding digit, e.g. `One` returns 1 and `Two` returns 2.

The `Conqueue` data types form the top-level conqueue structure – a conqueue is either a `Tip`, denoting the point where the two sequences of Conc-trees meet, or a `Spine`, denoting a pair of digits in the aforementioned number representation called `lwing` (left wing) and `rwing` (right wing), as well as the remaining conqueue `rear`. The reference `rear` to the remaining conqueue is a lazy value – it is only evaluated the first time it is accessed.

```
def inc[T](num: Num[T], c: Conc[T]) =
  num.rank match {
    case 0 =>
      One(c)
    case 1 =>
      val One(_1) = num
      Two(c, _1)
    case 2 =>
      val Two(_1, _2) = num
      Three(c, _1, _2)
    case _ =>
      error("Causes a carry.")
  }
```

```
def pushHead[T](conq: Conqueue[T], c: Conc[T]) =
  conq match {
    case Tip(tip) if tip.rank < 3 =>
      Tip(inc(tip, c))
    case Tip(Three(_1, _2, _3)) =>
      Spine(Two(c, _1), Two(_2, _3), Tip(Zero))
    case Spine(lw, rw, rear) if lw.rank == 3 =>
      Spine(inc(lw, c), rw, rear)
    case Spine(Three(_1, _2, _3), rw, rear) =>
      val nlw = Two(c, _1)
      val carry = _2 <> _3
      Spine(nlw, rw, $pushHead(rear, carry))
  }
```

Figure 3.8: Conqueue Push-Head Implementation

The Conqueue can check if its rear part was evaluated[2] with the evaluated method. The reason why we choose to make rear lazy will become apparent soon.

We will impose the following invariant on the conqueue data structure. A Spine that is k steps away from the root of the conqueue has a left and a right wing containing Conc-trees whose level is exactly k. The digit in the Tip that is k steps away from the root also contains Conc-trees with level k. With this invariant, adding an element into the conqueue corresponds to incrementing a number in the 4-digit base-2 number system. We call this the *rank invariant*.

Push-Head Operation

We show the implementation of the push-head operation in Figure 3.8. The pushHead operation relies on a helper method inc. This method appends a Conc-tree to a Num data type from the left, under the precondition that the Num node is less than Three. The pushHead operation checks the shape of the conqueue conq. If conq is a Tip with less than three Conc-trees, it simply calls the inc method. In the case of a Tip with three trees, it distributes the four Conc-trees into a spine with a Zero tip. This is shown in Figure 3.9 at the top of the leftmost column. If conq is a spine, we link the new tree c to the left wing lw, and leave the right wing rw intact. If necessary, a carry is created from two rightmost Conc-trees of the left wing and a new lazy rear is created for the new spine. We do not call pushHead recursively, as that could trigger a chain of carry operations and invalidate the $O(1)$ running time bound.

An astute reader will notice that this implementation is still not worst-case $O(1)$. Even though we do not invoke pushHead recursively, we do read the rear field when pushing. A series of accumulated lazy evaluation blocks could chain-react in the same way as a recursive call could. To prevent this from happening, we need to establish a well-defined

[2] Scala does not allow directly checking if the lazy value has been already evaluated, so our actual implementation encodes this differently. We assume that there is a method called isEvaluated on lazy values for simplicity. Furthermore, we will precede the expressions that initialize lazy values with a $ – in the absence of this notation we assume that the lazy value is forced to start with.

1	${}^{32}_{22}1$		${}^{2333333}_{2222222}0$
2	${}^{23}_{22}1$		${}^{3333333}_{2222222}0$
3	${}^{33}_{22}1$		${}^{2}_{2}\$^{433333}_{222222}0$
${}^{2}_{2}0$	${}^{2}_{2}\$^{4}_{2}1$		$\star{}^{22}_{22}\$^{43333}_{22222}0$
${}^{3}_{2}0$	$\star{}^{22}_{22}2$		${}^{32}_{22}\$^{43333}_{22222}0$
${}^{2}_{2}1$	${}^{32}_{22}2$		$\star{}^{322}_{222}\$^{4333}_{2222}0$
${}^{3}_{2}1$	${}^{23}_{22}2$		${}^{232}_{222}\$^{4333}_{2222}0$
${}^{2}_{2}2$	${}^{33}_{22}2$		$\star{}^{2322}_{2222}\$^{433}_{222}0$
${}^{3}_{2}2$	${}^{2}_{2}\$^{4}_{2}2$	$\cdots$	${}^{3322}_{2222}\$^{433}_{222}0$
${}^{2}_{2}3$	$\star{}^{22}_{22}3$		$\star{}^{33222}_{22222}\$^{43}_{22}0$
${}^{3}_{2}3$	${}^{32}_{22}3$		${}^{2}_{2}\$^{4222}_{2222}\$^{43}_{22}0$
${}^{22}_{22}0$	${}^{23}_{22}3$		$\star{}^{22322}_{22222}\$^{43}_{22}0$
${}^{32}_{22}0$	${}^{33}_{22}3$		${}^{32322}_{22222}\$^{43}_{22}0$
${}^{23}_{22}0$	${}^{2}_{2}\$^{4}_{2}3$		$\star{}^{323222}_{222222}\$^{4}_{2}0$
${}^{33}_{22}0$	$\star{}^{22}_{22}\$3$		${}^{233222}_{222222}\$^{4}_{2}0$
${}^{2}_{2}\$^{4}_{2}0$	${}^{222}_{222}0$		$\star{}^{2332222}_{2222222}1$
$\star{}^{22}_{22}1$	${}^{322}_{222}0$		${}^{3332222}_{2222222}1$

Figure 3.9: Lazy Conqueue Push-Head Illustration

schedule of evaluating the lazy blocks.

Figure 3.9 uses the 4-digit number system with base 2 to illustrate push-head operations in a conqueue. It uses the dollar sign $ followed by the digit 4 to denote the locations in this data structure where there is an unevaluated `rear`. Different entries in each column represent different conqueue states. Entries preceded by the ⋆ sign denote that a lazy `rear` was evaluated to reach this state. All other entries represent states immediately after a `pushHead` operation. In this evaluation schedule we choose to evaluate exactly one lazy `rear` after each `pushHead` operation (assuming there are any lazy `rear`s to evaluate). Note that at any point there might be more than one lazy `rear`, as is the case in the rightmost column. The important property of this evaluation schedule is that a dollar sign never directly precedes another dollar sign, so the evaluations can never chain-react.

To allow this evaluation schedule, we introduce the *lazy conqueues* with the data type `Lazy`, shown in Figure 3.10. This data type wraps a main `Conqueue q` and maintains two lists of references to `Spine`s in the main `Conqueue` that need to have their `rear`s evaluated. These lists are called `lstack` and `rstack` and they point to specific left and right wings in the conqueue. The new `lPushHead` method always evaluates at most one `rear` from the top of each stack.

```
class Lazy[T](
  lstack: List[Spine[T]],
  q: Conqueue[T],
  rstack: List[Spine[T]]
) extends Conqueue[T] {
  def left = q.left
  def right = q.right
  def level = q.level
  def size = q.size
  def evaluated = error()
  def rear = error()
  override def normalized =
    q.normalized
}

```

```
def lPushHead[T](lq: Lazy[T], s: Single[T]) = {
  val nq = pushHead(lq.q, s)
  val nlstack =
    if (nq.evaluated) pay(lstack, 1)
    else pay(nq :: lstack, 1)
  val nrstack = pay(rstack, 1)
  Lazy(nlstack, nq, nrstack)
}
def pay[T](work: List[Spine[T]], n: Int) =
  if (n == 0) work else work match {
    case head :: rest =>
      if (head.rear.evaluated) pay(rest, n - 1)
      else pay(head.rear :: rest)
    case Nil => Nil
  }
```

Figure 3.10: Lazy Conqueue Push-Head Implementation

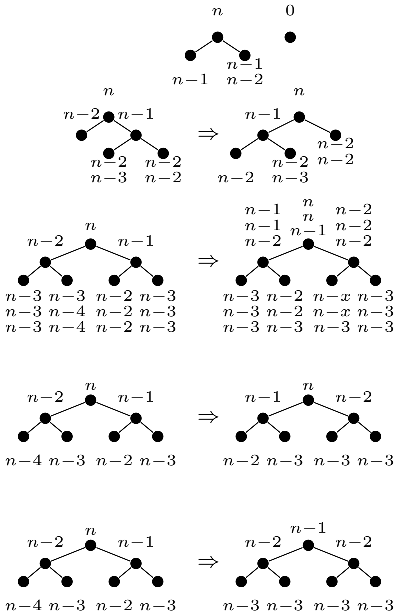
Tree Shaking

The pop-head operation will be similar to the push-head, with the difference that it will break trees rather than link them. In this sense, a popping an element closely corresponds to a decrement. There is one obstacle to implementing pop like this – while linking two trees of the same height guarantees increasing the tree height by one, breaking an arbitrary Conc-tree gives no guarantee that the resulting trees have the same height. Due to the balance invariant, a Conc-tree may be left-leaning, locally balanced or right-leaning. Breaking the Conc-tree of level n into two trees may give Conc-trees with levels $n - 1$ and $n - 2$, $n - 1$ and $n - 1$, or $n - 2$ and $n - 1$, respectively. We cannot swap the order of two trees in the conqueue, as that would change the ordering of elements.

It turns out that it is possible to control the Conc-tree levels after a break to a certain extent. In particular, it is possible to convert a left-leaning Conc-tree into either a locally balanced or a right-leaning Conc-tree, and similarly the right-leaning into either a locally balanced or a left-leaning one. We call this process *tree shaking*.

The `shakeLeft` operation shown in Figure 3.11 guarantees that the resulting tree is either left-leaning or locally balanced – the `shakeRight` method is its mirror image, so we do not show it. If the tree for the `shakeLeft` is not already in the right configuration, it and its subtrees need to be broken apart and relinked in a different order to yield the correct leaning. We show illustrations for different `shakeLeft` cases on the left in Figure 3.11. In each case we enumerate the possible subtree heights. For some of the cases, e.g. the third case, there are several different possibilities for the heights, so we stack them on top of each other. By enumerating and examining the possible configurations the Conc-tree can be in, we arrive at the following theorem.

Theorem 3.4 *The left-shake (right-shake) operation always returns a tree that is either left-leaning or locally balanced (respectively, right-leaning or locally balanced). If the height of the resulting tree is different than the input tree, then the resulting tree is locally balanced. The resulting tree is by at most* 1 *lower than the input tree.*



```
def shakeLeft[T](xs: Conc[T]): Conc[T] = {
  if (xs.level <= 1) xs
  else if (xs.left.level >= xs.right.level) xs
  else if (xs.right.right.level >= xs.right.left.level) {
    val nl = new <>(xs.left, xs.right.left)
    val nr = xs.right.right
    new <>(nl, nr)
  } else if (xs.left.left.level >= xs.left.right.level) {
    val nll = xs.left.left
    val nlr = new <>(xs.left.right, xs.right.left.left)
    val nl = new <>(nll, nlr)
    val nr = new <>(xs.right.left.right, xs.right.right)
    new <>(nl, nr)
  } else if (xs.right.left.left.level >=
    xs.right.left.right.level) {
    val nl = new <>(xs.left, xs.right.left.left)
    val nr = new <>(xs.right.left.right, xs.right.right)
    new <>(nl, nr)
  } else {
    val nll = new <>(xs.left.left, xs.left.right.left)
    val nlr =
      new <>(xs.left.right.right, xs.right.left.left)
    val nl = new <>(nll, nlr)
    val nr = new <>(xs.right.left.right, xs.right.right)
    new <>(nl, nr)
  }
}
```

Figure 3.11: Tree Shaking

Pop-Head Operation

As the discussion on tree shaking hints, popping from the conqueue does not exactly correspond to decrementing a 4-digit base-2 number. For example, the conqueue $^{102}_{222}0$ can yield an equivalent conqueue $^{211}_{222}0$, but it can also yield the conqueues $^{121}_{222}0$ and $^{11^\star 1}_{22\ 2}0$. However, whenever a borrow causes a Conc-tree to break into a single tree with a smaller height, we know that the next time that tree breaks it will yield two trees of the same height. This follows directly from Theorem 3.4 and we call such `One` trees *excited*, denoting them with $1^\star$.

The task of the `popHead` method is to never create two consequent, `Zero` or non-excited `One` trees. The `popHead` method needs to consider the following cases:

$$
\begin{array}{lllllllllll}
x & 0 & 2 & \Rightarrow & x & 1^* & 1 \qquad & x & 0 & y & \Rightarrow & x & 1^* & y_{-1} \\
x & 0 & 2 & \Rightarrow & x_{+1} & 2 & 0 \qquad & x & 0 & y & \Rightarrow & x_{+1} & 1 & y_{-1} \\
x & 0 & 2 & \Rightarrow & x & 3 & 0 \qquad & x & 0 & y & \Rightarrow & x & 2 & y_{-1}
\end{array}
$$

The pop-head operation seeks to maintain the following invariant – there is never an evaluated `rear` of some `Spine` node such that its wings are $^{x0z}_{pqr}$ or $^{xyz}_{p0r}$. We will call this the *no-zero invariant.* Additionally, it will ensure that there are never two consecutive non-excited `One` nodes, as this allows cascading. The special case `fuseAndBorrow` method fuses the `Num` node z (or r) and pulls it to a lower rank.

```
def dec[T](num: Num[T]) =
  num match {
    case Zero =>
      error("Requires a borrow.")
    case One(_) =>
      Zero
    case Two(_1, _2) =>
      One(_2)
    case Three(_1, _2, _3) =>
      Two(_2, _3)
  }

def popHead[T](conq: Conqueue[T]) = {
  conq match {
    case Tip(tip) =>
      Tip(dec(tip))
    case Spine(l, r, t) if l.rank > 1 =>
      Spine(dec(l), r, t)
    case Spine(One(_1), rw, r) =>
      r match {
        case Spine(rlw, rrw, rr) =>
          val rlm = rlw.leftmost
          val nlw =
            Two(rlm.left, rlm.right)
          val nrlw = dec(rlw)
          val nr = Spine(nrlw, rrw, rr)
          val ns = Spine(nlw, rw, nr)
          if (nrlw.rank > 0) ns
          else fix(ns)
        case Tip(Zero) =>
          Tip(rw)
        case Tip(tip) =>
          val lm = tip.leftmost
          val nlw =
            Two(lm.left, lm.right)
          val ntip = Tip(dec(tip))
          Spine(nlw, rw, ntip)
      }
  }

def fix[T](s: Spine[T]) = {
  def borrow(b: Conc[T], nrr: Conqueue[T]) = {
    val bs = shakeRight(b)
    if (bs.level == b.level) {
      if (bs.left.level == b.level - 1) {
        val nrlw = Two(bs.left, bs.right)
        val nr = Spine(nrlw, s.rear.rwing, nrr)
        Spine(s.lwing, s.rwing, $fix(nr))
      } else {
        val nrlw = One(bs.right)
        val nr = Spine(nrlw, s.rear.rwing, nrr)
        val nlw = incRight(s.lwing, bs.left)
        Spine(nlw, s.rwing, $fix(nr))
      }
    } else {
      val nrlw = One(bs)
      val nr = Spine(nrlw, s.rear.rwing, nrr)
      Spine(s.lwing, s.rwing, $fix(nr))
    }
  }
  s.rear match {
    case Spine(rlw, rrw, rr) if rlw.rank == 0 =>
      rr match {
        case Spine(rrlw, rrrw, rrr) =>
          if (doesNotCause11(rrlw)) {
            val rrlm = rrlw.leftmost
            val nrrlw = dec(rrlw)
            val nrr = Spine(nrrlw, rrrw, rrr)
            borrow(rrlm, nrr)
          } else fuseAndBorrow(s)
        case Tip(Zero) =>
          Spine(s.lwing, s.rwing, Tip(rrw))
        case Tip(tip) =>
          val nrr = Tip(dec(tip))
          borrow(tip.leftmost, nrr)
      }
    case _ =>
      s
  }
}
```

Figure 3.12: Conqueue Pop-Head Implementation

In Figure 3.12 we show the implementation of the pop-head operation. We break the `popHead` implementation into the base case that treats the top of the conqueue and assumes that the rank of the argument `conq` is 0. The more complex recursive `fix` method examines the conqueue two ranks deep and rewrites it accordingly with the `borrow` method. By the Theorem 3.4, a tree may break in subtrees in three different ways after shaking – the `borrow` method must treat these cases separately.

3.3.2 Normalization and Denormalization

Having shown that conqueues support $O(1)$ worst-case time deque operations, we turn our attention to concatenation. We recall from Section 3.1 that concatenation called `normalized` on its arguments. To show that `concat` is $O(\log n)$ for conqueues, it remains to show the implementation of the `wrap` method from Figure 3.7. This method is similar to the `wrap` method on Conc-tree ropes that we have seen earlier. We refer the reader to the (slightly longer) $O(\log n)$ implementation of `wrap` in our source code [Prokopec(2014a)], and show a simpler, $O(\log^2 n)$, version here:

```
def wrap[T](conq: Conqueue[T]): Conc[T] = {
  def wrap(lacc: Conc[T], q: Conqueue[T], racc: Conc[T]) = q match {
    case Spine(lw, rw, r) =>
      wrap(lacc <> lw.left <> lw.right, r, rw.left <> rw.right <> racc)
    case Tip(tip) =>
      lacc <> tip.left <> tip.right <> racc
  }
  wrap(Empty, conq, Empty)
}
```

However, the `concat` operation returns a Conc-tree and not a conqueue. To support concatenation for conqueues, we must have some means of denormalizing any Conc-tree back into a conqueue. This `denormalized` operation does the opposite of the `wrap` operation – it unwinds smaller and smaller Conc-trees until it ends up with two long lists. It consists of two tail recursive methods `unwrap` and `zip`. The `unwrap` method maintains two stacks of `Num` nodes for the left and the right wings and the remaining middle of the Conc-tree. It continuously prunes the middle of the Conc-tree and adds Conc-trees to the smaller stack of `Num` nodes to ensure there are as many left wings as there are right wings. Once this is done, the `zip` method simply zips the wings together. We show the `denormalized` implementation in Figure 3.13 – its added complexity confirms the folklore that it is *easier to break things than to build them.*

3.4 Conc-Tree Combiners

Most of the data structures shown so far were immutable. Although parts of the lazy conqueues were modified after already being constructed, the semantics of lazy values ensured that a specific lazy conqueue value is always observationally the same. The data structures so far can thus all be called *persistent.*

However, this persistence comes at a cost – while adding a single node to the data structure has an $O(1)$ running time, the constant factors involved are still large. We have seen how introducing `Chunk` nodes can amortize these costs in Section 3.2. In this section we expand this idea and show several mutable data structure that use conc-ropes as basic building blocks.

3.4.1 Conc-Tree Array Combiner

Although the `ArrayCombiner` shown in Section 2.4 has $O(1)$ appends, its resizing strategy requires it to on average write every element twice to memory. Additionally, in a managed runtime like the JVM reallocating an array requires a sweep phase that zeroes out its values, only to have the combiner rewrite those default values afterwards. All these computations add to the constant factors of the `+=` operation.

```
def denormalized[T](xs: <>[T]): Conqueue[T] = {
  def unwrap(lstack: List[Num[T]], rstack: List[Num[T]], rem: Conqueue[Conc[T]]) = {
    if (rem.isEmpty) (lstack.reverse, rstack.reverse)
    else if (lstack.length < rstack.length) {
      val remhead = rem.head
      if (
        (lstack.nonEmpty && lstack.head.rightmost.level < remhead.level) ||
        (lstack.isEmpty && remhead.level > 0)
      ) {
        val nrem = remhead.left +: remhead.right +: rem.tail
        unwrap(lstack, rstack, nrem)
      } else (lstack: @unchecked) match {
        case Three(_1, _2, _3) :: ltail =>
          val added = _3 <> remhead
          if (added.level == _3.level) unwrap(Three(_1, _2, added) :: ltail, rstack, rem.tail)
          else unwrap(One(added) :: Two(_1, _2) :: ltail, rstack, rem.tail)
        case Two(_1, _2) :: ltail =>
          val added = _2 <> remhead
          if (added.level == _2.level) unwrap(Two(_1, added) :: ltail, rstack, rem.tail)
          else unwrap(One(added) :: One(_1) :: ltail, rstack, rem.tail)
        case One(_1) :: Nil =>
          val added = _1 <> remhead
          unwrap(Two(added.left, added.right) :: Nil, rstack, rem.tail)
        case One(_1) :: num :: ltail =>
          val added = _1 <> remhead
          val shaken = if (added.level == _1.level) added else shakeRight(added)
          if (shaken.level == _1.level) {
            unwrap(One(shaken) :: num :: ltail, rstack, rem.tail)
          } else if (shaken.left.level == shaken.right.level) {
            unwrap(Two(shaken.left, shaken.right) :: num :: ltail, rstack, rem.tail)
          } else num match {
            case Three(n1, n2, n3) =>
              unwrap(Two(n3 <> shaken.left, shaken.right) :: Two(n1, n2) :: ltail,
                rstack, rem.tail)
            case num =>
              unwrap(One(shaken.right) :: incRight(num, shaken.left) :: ltail,
                rstack, rem.tail)
          }
        case Nil =>
          unwrap(One(remhead) :: Nil, rstack, rem.tail)
      }
    } else {
      // the lstack.length >= rstack.length case is mirrored
    }
  }

  def zip(rank: Int, lstack: List[Num[T]], rstack: List[Num[T]]): Conqueue[T] =
    (lstack, rstack) match {
      case (lwing :: Nil, Nil) =>
        Tip(lwing)
      case (Nil, rwing :: Nil) =>
        Tip(rwing)
      case (lwing :: Nil, rwing :: Nil) =>
        new Spine(lwing, rwing, Tip(Zero))
      case (lwing :: ltail, rwing :: rtail) =>
        new Spine(lwing, rwing, zip(rank + 1, ltail, rtail))
    }

  val (lwings, rwings) = unwrap(Nil, Nil, Tip(One(new Single(xs))))
  zip(0, lwings, rwings)
}
```

Figure 3.13: Conqueue Denormalization

We recall that the two-step array combiner adds the elements once, merges with different combiners and then efficiently traverses all the elements. Since most of the time a data-parallel transformer operation either adds elements to the combiner or traverses them, we should decrease the constant factors associated with `+=`. We analyze its efficiency in terms of the number of writes to memory.

```
class ConcBuffer[T](val k: Int) {
  private var conc: Conc[T] = Empty
  private var chunk: Array[T] = new Array(k)
  private var lastSize: Int = 0

  def +=(elem: T): this.type = {
    if (lastSize >= k) expand()
    chunk(lastSize) = elem
    lastSize += 1
    this
  }

  private def pack() {
    conc = append(conc, new Chunk(chunk, lastSize, k))
  }

  private def expand() {
    pack()
    chunk = new Array(k)
    lastSize = 0
  }
}
```

Conc-ropes with `Chunk` leaves ensure that every element is written only once. The larger the maximum chunk size `k` is, the less often is a `Conc` operation invoked in the method `pack` – this amortizes conc-rope append cost, while giving the benefits of fast traversal. The `ConcBuffer` shown above is much faster than the `ArrayBuffer` when streaming in elements, while in the same time supporting efficient concatenation. Due to the underlying immutable conc-rope this buffer also allows efficient copy-on-write snapshot operations.

3.4.2 Conc-Tree Hash Combiner

The two-step hash combiners shown in Section 2.4 have the same resizing issues as array combiners. We modify the `HashCombiner` to have each bucket implemented as a Conc-tree array combiner. Appending to the `HashCombiner` thus becomes very efficient in terms of constant factors, and combining two `HashCombiners` becomes an $O(P \log n)$ operation.

3.5 Related Work

Standard library collection packages of most languages come with resizeable array implementations, e.g. the `ArrayList` in the JDK or the `vector` in C++ standard template library. These are mutable data structures that provide $O(1)$ worst case time indexing and update operations, with $O(1)$ amortized time append operation. Their concatenation is an $O(n)$ operation. Functional languages come with functional cons-lists that allow efficient prepend and pop operations, but most other operations are $O(n)$.

Ropes are heavily relied upon in the Xerox Cedar environment. Their description by Boehm, Atkinson and Plass [Boehm et al.(1995)Boehm, Atkinson, and Plass] performs bulk rebalancing, after the rope becomes particularly skewed. These ropes guarantee amortized, but not worst-case, $O(\log n)$ complexity. VList [Bagwell(2002)] is a functional data structure for storing sequences, with logarithmic time lookup operations. Scala Vectors [Bagwell and Rompf(2011)] are based on the immutable sequence implementation. Its deque operations have a low constant factor, but require $O(\log n)$ time. Compared to the standard implementation, conc-ropes have more efficient appends and are thus more suited as combiners. The standard Vector implementation does not support concatentation, since adding concatenation slows down the append and other operations.

Fortress expresses parallelism using recursion and pattern matching on three node types [Allen et al.(2007)Allen, Chase, Hallett, Luchangco, Maessen, Ryu, Jr., and Tobin]. All Conc-tree variants in this chapter provide the same programming model as Fortress Conc lists [Steele(2009)], and this chapter investigates how to efficiently implement Conc list operations.

Relaxing the balancing requirements to allow efficient updates was first proposed for a data structure called the AVL tree [Adelson-Velsky and Landis(1962)]. The recursive slowdown techniques were first introduced by Kaplan and Tarjan [Kaplan and Tarjan(1995)]. Okasaki was one of the first researchers to bridge the gap between amortization and persistence through the use of lazy evaluation [Okasaki(1996)].

Okasaki [Okasaki(1998)] gives a good overview and an introduction to the field of persistent data structures. The catenable real-time queues due to Okasaki allow efficient concatenation but do not have the balanced tree structure and are thus not suitable for parallelization, nor they support logarithmic random access [Okasaki(1997)]. Hinze and Paterson describe an implementation of a lazy finger tree data structure [Hinze and Paterson(2006)] with *amortized* constant time deque and concatenation operations.

3.6 Conclusion

This chapter introduced Conc-tree data structures. We learned that different Conc-tree variants are suitable for a range of different tasks. The basic Conc-tree list comes with a worst-case $O(\log \frac{n_1}{n_2})$ time concatenation operation with a low constant factor, and can be used to implement Fortress-style Conc-tree lists [Steele(2009)]. The Conc-tree rope provides an amortized $O(1)$ time ephemeral append operation, while retaining the worst-case $O(\log n)$ time concatenation bound. By introducing `Chunk` nodes to improve memory footprint and data locality, the Conc-tree rope becomes an ideal data structure for data-parallel *combiners* [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky]. Finally, conqueue supports worst-case $O(1)$ persistent deque operations along with worst-case $O(\log n)$ time persistent split and concatenation, making it a good finger tree implementation for both functional and imperative languages. While conqueues retain the same asymptotic running time bounds as simpler Conc-tree variants, they have somewhat larger constant factors due to extra complexity.

Which of these data structures should we use in practice? Along with the traditional wisdom of picking the proper asymptotic complexity, this choice should be driven by pragmatism. We should only pay for the extra implementation complexity of conqueues in real-time applications, or those functional programs in which the persistent use of a data structure is critical. For functional programs in which ephemeral use suffices, and non-real-time applications, the Conc-tree ropes are a better choice, both in terms of better constant factors and their simplicity.

4 Parallelizing Reactive and Concurrent Data Structures

In Chapter 2 we made several assumptions about the way data structures are used. First of all, we assumed that operations execute in bulk-synchronous mode. Second, we assumed that all the elements in the collection are present before a data-parallel operation is invoked. Finally, we assumed quiescence during the execution of the operation. In this chapter we will drop some of these constraints. In Section 4.1, we will not require that the elements are present in the data structure before invoking the operation, and study two examples of *reactive parallelization*.

In Section 4.2 we will study how data structure operations can be parallelized efficiently without the quiescence assumption. In doing so, we will rely on a lock-free, linearizable, lazy *snapshot* operation. Analogous to how lazy evaluation allows applying amortization techniques to persistent data structures [Okasaki(1998)], in Section 4.2, we will find that adding lazyness to concurrent data structures allows applying scalable parallel snapshot techniques. This surprising duality comes from the fact that, while lazyness allows separate persistent data structure operations to share work, concurrent data structure operations rely on lazyness to execute parts of the work in isolation.

4.1 Reactive Parallelization

There are many applications in which an operation needs to be performed on data elements that arrive later. Streaming is one such example. We want to be able to express what happens to the data although the data arrives from some source much later. Another example is an asynchronous computation. Result from a potentially long-running asynchronous operation is added to a data structure long after specifying what to do with that result.

Dropping the assumption about the presence of elements has several consequences. The fact that the elements are not present when the operation begins means that they have to be added by some other thread. The fact that the data structure must be concurrently modified during the execution of the operation further implies that we must drop the quiescence assumption as well – we already noted that quiescence implies the presence of elements, so this follows naturally. Furthermore, we will focus on non-blocking data structure operations. Since we do not know when more elements will become available, we do not want to block the thread calling the operation, so we drop the bulk synchronous execution assumption as well.

We call this type of parallelization *reactive*, because the operation reacts to the addition of new elements into the data structure. It is not known when the elements will arrive, but when they do, a certain operation needs to be executed on them. In the context of this section, we focus on reactive data structures that result in deterministic programming models.

To illustrate these concepts we start by introducing the simplest possible such data structure, which can hold at most one element. This data structure is additionally restricted by not allowing the contained element to change after it is added. We call this data structure a *future* or a *single-assignment variable*.

4.1.1 Futures – Single-Assignment Values

The *single-assignment variable* is a data structure that can be assigned to only once. Before it is assigned it contains no data elements – we say that it is empty or unassigned. After it is assigned an element, it is never mutated again – trying to reassign it is a program error. This error may manifest itself at compile-time or at runtime, and for simplicity we will assume the latter[1]. A single assignment variable may be read at any point. Reading it before it has been assigned is either a runtime error, results in an invalid value, or postpones the read until the data element is available. Of the three alternatives we assume the last one – this ensures that multithreaded programs written using single-assignment variables are deterministic. Postponing could either block the reading thread or install a callback that is called later. In the interest of achieving better throughput we choose the latter.

In this section we divide the single-assignment variables into two separate abstractions – the *future* end that allows reading its value and the *promise* end that allows assigning a value [Haller et al.(2012)Haller, Prokopec, Miller, Klang, Kuhn, and Jovanovic]. Every future corresponds to exactly one promise and vice versa. In the implementation these will actually comprise the same data structure in memory, exposed through two different interfaces.

In Figure 4.1 we show two data types `Future` and a `Promise`[2]. We call the methods that

[1] Compile-time double assignment checks can only be approximated and would restrict the programming model further.

[2] For simplicity we omit the fact that the Scala `Promise`s can be completed with exceptions that denote

```
trait Future[T] {
  def onSuccess(f: T => Unit): Unit
}
```

```
trait Promise[T] {
  def success(elem: T): Unit
  def future: Future[T]
}
```

Figure 4.1: Futures and Promises Data Types

write and read the element `success` and `onSuccess`, respectively. Every `Promise[T]` can return a reference to its corresponding `Future` with the `future` method. The `Future[T]` type does not have a method for obtaining the corresponding `Promise[T]`, as such design would allow consumers to write to the future.

Using the `task` statement from Chapter 2 futures and promises can be used to implement an asynchronous computation statement as follows:

```
def future[T](body: =>T): Future[T] = {
  val p = newPromise
  task {
    p.success(body)
  }
  p.future
}
```

The future object returned by this statement is eventually completed with the result of the `body` computation. A consumer can subscribe to the result using the `onSuccess` method, executing a side-effect such as completing another future. This side-effect will be executed eventually, but only after the future is completed with a value. In the example below we show how the result of the future can be used to fetch a url using the `http` protocol and another using the `https` protocol. The second example cannot display the obtained value in the browser directly, but must `decrypt` them first.

```
val f = future { http(url) }
f onSuccess {
  html => gui.text = html
}
```

```
val f = future { https(url) } map { decrypt }
f onSuccess {
  html => gui.text = html
}
```

This decryption is done using the `map` combinator on futures, Similar to the `map` combinator already seen on collections. Its implementation is almost identical to the `map` implementation for sequential collections.

errors in asynchronous computations.

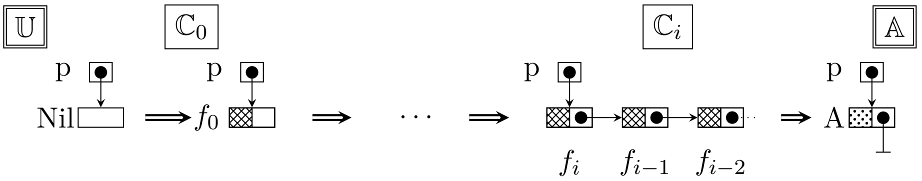


Figure 4.2: Future States

```
def map[S](f: T => S): Future[S] = {
  val p = newPromise
  this onSuccess {
    elem => p.success(f(elem))
  }
  p.future
}
```

Replacing newPromise, onSuccess and success with newBuilder, foreach and += shows that, despite the important semantic differences in these operations, a map implementation is identical to the one seen earlier. Thus, a Promise can be regarded as a Builder for the Future, and its onSuccess method can be regarded as its foreach. We can define a number of other functional combinators such as flatMap, filter and zip in a similar way.

Implementation

It remains to show a typical future implementation. We note that the future can be in several states. First, it is in the uninitialized state after it is created ($\mathbb{U}$). Then, callbacks may be added to the future – each addition of a callback represents a different state ($\mathbb{C}_i$). Finally, after it is assigned, it is in the assigned state ($\mathbb{A}$). The flow between these states is well defined and no state can be entered more than once. We show the flow in Figure 4.2. This useful property ensures that the implementation in this section will not suffer from the ABA problem.

Due to the simplicity of its state diagram along with the fact that there is at most a single element that can logically reside in a Future, the future implementation serves as a textbook concurrent data structure example. The future data structure will occupy a single location in memory globally visible to all the threads and manipulated by CAS instructions. Figure 4.2 illustrates the states of this single memory location p. When unassigned, this memory location will point to an empty list Nil. To add a callback is to atomically replace the current list with an updated list of callbacks f_i. Once the element is finally assigned, we replace the callbacks with the value of the element. After that the callbacks need not be added for later execution anymore, but can be executed directly with the value of the element.

```
class SAV[T] extends Promise[T] with Future[T] {
  atomic var p = Nil

  def onSuccess(f: T => Unit) = {
    val state = READ(p)
    state match {
      case cs: Callbacks =>
        if (!CAS(p, cs, f :: cs)) onSuccess(f)
      case elem =>
        f(elem)
    }
  }

  def future = this

  def success(elem: T) = {
    val state = READ(p)
    state match {
      case cs: Callbacks =>
        if (!CAS(p, cs, elem)) success(elem)
        else for (f <- cs) f(elem)
      case elem =>
        error("Already assigned.")
    }
  }
}
```

Figure 4.3: Future Implementation

We show the complete implementation in Figure 4.3. Note that the implementation, as most lock-free algorithms we will show, uses tail-recursion. An unsuccessful CAS instruction simply restarts the subroutine. The implementation is lock-free – a failed CAS can only mean that another thread succeeded in installing a callback or assigning the value. The monotonicity of the future's state diagram ensures that the CASes succeed if and only if there was no change between the last read and the CAS. The changes are thus atomic, and it is trivial to see that the implementation is correct.

4.1.2 FlowPools – Single-Assignment Pools

In this section, we describe a data type called a `FlowPool`, together with its implementation [Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky]. As we will see, `FlowPool`s are prone to reactive parallelization and, as an added benefit, programs built using `FlowPool`s have deterministic semantics, as was the case with `Future`s described in the previous section.

`FlowPool`s are a generalization of `Future`s in the sense that they allow multiple values to be assigned. While a callback registered to a `Future` is called only once for the value of the future, a callback registered to a `FlowPool` is called for every value assigned to the `FlowPool`. This allows some new combinator methods that were not previously defined on `Future`s. Besides a richer programming model, `FlowPool`s process values in bulk, which allows higher scalability.

We start by describing the `FlowPool` abstract data type and its operations in more detail. We then show how basic `FlowPool` operations are used to implement more expressive combinators. Finally, we present a concrete lock-free `FlowPool` implementation that aims to achieve a low memory footprint. The specific implementation that we will study is not limited to the `FlowPool` abstract data type – it can easily be modified to serve as a concurrent queue implementation.

Programming Model

FlowPools define four basic operations [3]. The first operation is called create – it creates a FlowPool instance. The state-changing append operation (+=) adds values to the FlowPool. FlowPool traversal registers a callback that is called every time a value is added to the FlowPool. These three operations are analogous to the ones already seen on Futures and Promises. The fourth operation seal disallows further writes to the FlowPool after it reaches a certain size. Note that for a Promise this value is always 1 because it can only be assigned a single value – subsequent writes are implicitly disallowed after calling the success method, so there is no need for a seal operation on Future. Although we can make a distinction between the producer and consumer interfaces [Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky] as we did with Futures and Promises, we will avoid it for simplicity.

Create. This operation simply creates a fresh FlowPool instance. A FlowPool is initially empty and does not contain any values. Upon creation there is no bound on the number of values that can be added – we say that the FlowPool is not sealed.

```
def create[T](): FlowPool[T]
```

Append (+=). Given a FlowPool with a set of values, appending changes its state so that it contains the specified value. This method is thread-safe and linearizable. This is the only operation that changes the state of the FlowPool.

We choose the following signature to allow chaining append calls:

```
def append(v: T): FlowPool[T]
```

Foreach and Aggregate. A abstraction that represents a set of values is only useful if those values can be accessed in one way or another. Values cannot be queried for their presence in a FlowPool, just as it is not possible to directly check if a Future contains a value – if this were allowed, the determinism in the programs using FlowPools would be compromised. Assume FlowPools had an operation contains that checks whether a given value is contained. The following program with two threads and contains is non-deterministic in the sense that different execution schedules produce different outputs:

```
val pool = create()
thread {
  if (pool.contains(1)) println("Found.")
}
pool += 1
```

[3] A slightly more involved programming model also defines builders that ensure that unneeded values are properly garbage collected [Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky].

The program creates a fresh `FlowPool` named `pool` and asynchronously starts another thread. It then adds a single value `1` to the `pool`. The asynchronously running thread uses `contains` to check if the value `1` is in the pool or not and prints a message `"Found."` based on that. Depending on whether the main program or the newly created thread runs faster, the output of the program is different.

Instead of providing query operations on a specific values which may or may not be present at a specific time index, a `FlowPool` defines operations that visit all its values – namely, traversal operations.

Typically, traversal is provided through iterators whose state may be manipulated by several threads, which could also violate determinism. Another way to traverse the elements is to provide a higher-order `foreach` operator which takes a callback argument and applies it to each element. Again, determinism is ensured by calling `foreach` on every value that is eventually added to the `FlowPool`, instead of only the values present in the `FlowPool` at the time it was created. Values can be added to the `FlowPool` as long as the limit set by `seal` is not reached, so a synchronous `foreach` seen on traditional collection types would have to block the caller thread. For this reason the `foreach` is asynchronous as it was on `Future`s – invoking it installs a callback, which is called later.

```
def foreach[U](f: T => U): Future[Int]
```

We choose return type of `foreach` to be a `Future` that becomes available once all the elements have been traversed. The value of the future is set to the number of times that `f` has been called for a value.

While the `foreach` operation is analogous to the one seen earlier on `Future`s, it is not powerful enough to express certain kinds of programs. The `Future` resulting from `foreach` does not contain enough useful information about the traversal.

Imagine we have a `FlowPool` of integer values (e.g. purchases) and we want to find their sum. The only way to do this with the operations described so far is by using a mutable accumulation variable:

```
def sum(pool: FlowPool[Int]): Future[Int] = {
  val p = Promise[Int]()
  var s = 0
  for (v <- pool) {
    s += v
  } onComplete { p.success(s) }
  p.future
}
```

Using only `Future`s and the `FlowPool` operations seen so far it is not possible to express this program. In fact, the solution using a mutable accumulator is not even correct. First of all, the variable s is not annotated as thread-safe (e.g. with a @volatile annotation). Second, this solution is susceptible to race conditions. More importantly, the callback installed with a foreach call is invoked eventually for every element, just as is the case with Futures. This means that s may be modified after the Future returned by foreach completes and the promise p is assigned.

The aggregate operation is a more expressive FlowPool primitive that aggregates the elements of the pool as the name implies. It has the following signature:

```
def aggregate[S](zero: =>S)(cb: (S, S) => S)(op: (S, T) => S): Future[S]
```

where zero is the initial aggregation, cb is an associative operator which combines several aggregations, op is an operator that adds an element to the aggregation, and Future[S] is the final aggregation of all the elements, which becomes available once all the elements have been added. Using aggregate, the previous sum subroutine can be expressed as follows:

```
def sum(pool: FlowPool[Int]): Future[Int] =
  pool.aggregate(0)(_ + _)(_ + _)
```

The aggregate operator also generalizes foreach:

```
def foreach[U](f: T => U): Future[Int] =
  pool.aggregate(0)(_ + _) { (no, x) =>
    f(x)
    no + 1
  }
```

The aggregate operator can divide elements into subsets, apply the aggregation operator op to aggregate elements in each subset starting from the zero aggregation, and then combine different subset aggregations by applying the cb operator to them. Alternative implementations [Schlatter et al.(2012)Schlatter, Prokopec, Miller, Haller, and Odersky] can guarantee horizontal scalability by evaluating these subsets in parallel. In essence, the first part of aggregate defines the commutative monoid and the functions involved must be non-side-effecting. In contrast, the operator op is guaranteed to be called only once per element and it can have side-effects.

While in an imperative programming model foreach and aggregate are equivalent in the sense that one can be implemented in terms of the other, in a single-assignment programming model aggregate is more expressive than foreach. The foreach operation can be implemented using aggregate, but not vice versa, due to the absence of mutable variables.

Importantly, note that FlowPools have no ordering between their values. Callbacks may be called for the values in an order that is different than the order in which those values were added to the pool. Specific `FlowPool` implementations may guarantee some form of ordering.

Seal. Once no more values are added to the pool or the bound on the number of elements is established, further appends can be disallowed by calling the `seal` operation. This allows `aggregate` to complete its future since it has a guarantee that no more elements will be added.

```
def seal(size: Int): Unit
```

A variant of `seal` that simply prevents appends the FlowPool at the moment it is called without specifying the size yields a nondeterministic programming model. This is illustrated by the following example program:

```
val pool = create()
thread {
  pool.seal()
}
pool += 1
```

A thread is attempting to seal the pool by executing concurrently with a thread that appends an element. In one execution, the append can precede the seal, and in the other the append can follow the seal, causing an error. To avoid nondeterminism, there has to be an agreement on the final state of the pool. A sufficient way to make `seal` deterministic is to provide the expected pool size as an argument. The semantics of `seal` is that it fails if the pool is already sealed with a different size or the number of elements is greater than the desired size.

Higher-order operators

The described basic operations are used as building blocks for other `FlowPool` operations. To formalize the notion that `FlowPool`s are generalized `Future`s, we note that a `FlowPool` can be used to implement a pair of a `Promise` and a `Future` as follows:

```
def promiseFuture[T](): (T => Unit, (T => Unit) => Unit) = {
  val pool = create[T]()
  pool.seal(1)
  (v => pool += v, f => pool.foreach(f))
}
```

In addition to the default `FlowPool` constructor, it is useful to have factory methods that create certain pools. In a dataflow graph, FlowPools created by these factory methods are represented as source nodes. The `iterate` factory method, which creates a `FlowPool`

```
def map[S](f: T => S) = {
  val p = new FlowPool[S]
  for (x <- this) {
    p += f(x)
  } map {
    sz => p.seal(sz)
  }
  p
}
```

```
def filter(p: T => Boolean) = {
  val fp = new FlowPool[T]
  aggregate(0)(_ + _) {
    (acc, x) => if p(x) {
      fp += x; acc + 1
    } else acc
  } map { sz => fp.seal(sz) }
  fp
}
```

```
def flatMap[S]
  (f: T => FlowPool[S]) = {
  val p = new FlowPool[S]
  aggregate(future(0))(add) {
    (af, x) =>
    val sf = for (y <- f(x)) p += y
    for (a <- af; s <- sf)
      yield a + s
  } map { sz => b.seal(sz) }
  p
}
```

Figure 4.4: FlowPool Monadic Operations

containing an infinite set of values `s`, `f(s)`, `f(f(s))`, `...`, is shown below. The `iterate` factory method creates a `FlowPool` of an arbitrary type `T`. It then starts an asynchronous computation using the `future` construct seen earlier, which recursively applies `f` to each number and adds it to the builder. Finally, a reference to the pool is returned. More complex factory methods that add values from a network socket or a database are also possible.

```
def iterate[T](s: T, f: T => T): FlowPool[T] = {
  val p = new FlowPool[T]
  def recurse(x: T) {
    p << x
    recurse(f(x))
  }
  future { recurse(s) }
  p
}
```

The operations used in Scala `for`-comprehensions are shown in Figure 4.4. The higher-order `map` operation maps each value of the `FlowPool` to produce a new one. This corresponds to chaining the nodes in a dataflow graph. We implement `map` by traversing the values of the `this` FlowPool and appending each mapped value. Once all the values have been mapped and there are no more values to traverse, we can safely seal the resulting `FlowPool`.

The `filter` operation produces a new `FlowPool` containing only the elements for which a specified predicate `p` holds. Appending the elements to a new pool works as with the `map` operation, but the seal needs to know the exact number of elements added. The `aggregate` accumulator is thus used to track the number of added elements.

The `flatMap` operation retrieves a separate `FlowPool` from each value of `this` pool and appends its elements to the resulting pool. The implementation is similar to that of `filter`, but the resulting `FlowPool` size is folded over the future values of intermediate pools. This is because intermediate pools possibly have an unbound size. The `flatMap` operation corresponds to joining nodes in the dataflow graph.

```
type Terminal {
  sealed: Int
  callbacks: List[Elem => Unit]
}

type Elem
```

```
type Block {
  array: Array[Elem]
  next: Block
  index: Int
  blockindex: Int
}
```

```
type FlowPool {
  start: Block
  current: Block
}
LASTELEMPOS = BLOCKSIZE - 2
NOSEAL = -1
```

Figure 4.5: FlowPool Data Types

```
def create()
  new FlowPool {
    start = createBlock(0)
    current = start
  }

```

```
def createBlock(bidx: Int)
  new Block {
    array = new Array(BLOCKSIZE)
    index = 0
    blockindex = bidx
    next = null }
```

Figure 4.6: FlowPool Creation

The `union` operation can be implemented in a similar manner. Note that if we could somehow merge the two different `foreach` operations to implement the third join type `zip`, we would obtain a nondeterministic operation. The programming model does not allow us to do so. The `zip` function is better suited for data structures with deterministic ordering, such as Oz streams [Roy and Haridi(2004)].

Implementation

One straightforward way to implement a growing pool is to use a linked list of nodes that wrap elements. As we are concerned about the memory footprint and cache-locality, we store the elements into arrays instead, which we call blocks. Whenever a block becomes full, a new block is allocated and the previous block is made to point to the `next` block. This way, most writes amount to a simple array-write, while allocation occurs only occasionally. Each block contains a hint `index` to the first free entry in the array, i.e. one that does not contain an element. An `index` is a hint, since it may actually reference an earlier index. The FlowPool maintains a reference to the first block called `start`. It also maintains a hint to the last block in the chain of blocks, called `current`. This reference may not always be up-to-date, but it always points to some block in the chain.

Each FlowPool is associated with a list of callbacks which have to be called in the future as new elements are added. Each FlowPool can be in a sealed state, meaning there is a bound on the number of elements it stores. This information is stored as a `Terminal` value in the first free entry of the array. At all times we maintain the invariant that the array in each block starts with a sequence of elements, followed by a `Terminal` delimiter. From a higher-level perspective, appending an element starts by copying the `Terminal` value to the next entry and then overwriting the current entry with the element being appended.

The `append` operation starts by reading the `current` block and the `index` of the free

```
13 def append(elem: Elem)
14   b = READ(current)
15   idx = READ(b.index)
16   nexto = READ(b.array(idx + 1))
17   curo = READ(b.array(idx))
18   if check(b, idx, curo) {
19     if CAS(b.array(idx + 1), nexto, curo) {
20       if CAS(b.array(idx), curo, elem) {
21         WRITE(b.index, idx + 1)
22         invokeCallbacks(elem, curo)
23       } else append(elem)
24     } else append(elem)
25   } else {
26     advance()
27     append(elem)
28   }
29
30 def advance()
31   b = READ(current)
32   idx = READ(b.index)
33   if idx > LASTELEMPOS
34     expand(b, b.array(idx))
35   else {
36     obj = READ(b.array(idx))
37     if obj is Elem WRITE(b.index, idx + 1)
38   }
```

```
39 def expand(b: Block, t: Terminal)
40   nb = READ(b.next)
41   if nb is null {
42     nb = createBlock(b.blockindex + 1)
43     nb.array(0) = t
44     if CAS(b.next, null, nb)
45       expand(b, t)
46   } else {
47     CAS(current, b, nb)
48   }
49
50 def check(b: Block, idx: Int, curo: Object)
51   if idx > LASTELEMPOS return false
52   else curo match {
53     elem: Elem =>
54       return false
55     term: Terminal =>
56       if term.sealed = NOSEAL return true
57       else {
58         if totalElems(b, idx) < term.sealed
59           return true
60         else error("sealed")
61       }
62     null =>
63       error("unreachable")
64   }
```

Figure 4.7: FlowPool Append Operation

position. It then reads the `nexto` after the first free entry, followed by a read of the `curo` at the free entry. The `check` procedure checks the bounds conditions, whether the FlowPool was already sealed or if the current array entry contains an element. In either of these events, the `current` and `index` values need to be set – this is done in the `advance` procedure. We call this the **slow path** of the `append` method. Notice that there are several causes that trigger the slow path. If some other thread completes the `append` method but is preempted before updating the value of the hint `index`, then the `curo` will have the type `Elem`. The same happens if a preempted thread updates the value of the hint `index` after additional elements have been added, via unconditional write in line 21. Finally, reaching an end of block triggers the slow path.

Otherwise, the operation executes the **fast path** and appends an element. It first copies the `Terminal` value to the next entry with a CAS instruction in line 19, with `nexto` being the expected value. If it fails (e.g. due to a concurrent CAS), the append operation is restarted. Otherwise, it proceeds by writing the element to the current entry with a CAS in line 20, the expected value being `curo`. On success it updates the `b.index` value and invokes all the callbacks (present when the element was added) with the `future` construct. In the implementation we do not schedule an asynchronous computation for each element. Instead, the callback invocations are batched to avoid the scheduling overhead – the array is scanned for new elements until there are no more left.

Interestingly, inverting the order of the reads in lines 16 and 17 would cause a race in which a thread could overwrite a `Terminal` value with some older `Terminal` value if

```
def seal(size: Int)
  b = READ(current)
  idx = READ(b.index)
  if idx <= LASTELEMPOS {
    curo = READ(b.array(idx))
    curo match {
      term: Terminal =>
        if ¬tryWriteSeal(term, b, idx, size)
          seal(size)
      elem: Elem =>
        WRITE(b.index, idx + 1)
        seal(size)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(idx))
    seal(size)
  }
def tryWriteSeal(term: Terminal, b: Block,
  idx: Int, size: Int)
  val total = totalElems(b, idx)
  if total > size error("too many elements")
  if term.sealed = NOSEAL {
    nterm = new Terminal {
      sealed = size
      callbacks = term.callbacks
    }
    return CAS(b.array(idx), term, nterm)
  } else if term.sealed ≠ size {
    error("sealed different size")
  } else return true
def totalElems(b: Block, idx: Int)
  return b.blockindex * (BLOCKSIZE - 1) + idx

def invokeCallbacks(e: Elem, term: Terminal)
  for (f <- term.callbacks) future {
    f(e)
  }
def asyncFor(f: Elem => Unit, b: Block, i: Int)
  if i <= LASTELEMPOS {
    obj = READ(b.array(i))
    obj match {
      term: Terminal =>
        nterm = new Terminal {
          sealed = term.sealed
          callbacks = f ∪ term.callbacks
        }
        if ¬CAS(b.array(i), term, nterm)
          asyncFor(f, b, i)
      elem: Elem =>
        f(elem)
        asyncFor(f, b, i + 1)
      null =>
        error("unreachable")
    }
  } else {
    expand(b, b.array(i))
    asyncFor(f, b.next, 0)
  }
def foreach(f: Elem => Unit)
  future {
    asyncFor(f, start, 0)
  }
```

Figure 4.8: FlowPool Seal and Foreach Operations

some other thread appended an element in between.

The `seal` operation continuously increases the `index` in the block until it finds the first free entry. It then tries to replace the `Terminal` value there with a new `Terminal` value which has the seal size set. An error occurs if a different seal size is set already. The `foreach` operation works in a similar way, but is executed asynchronously. Unlike `seal`, it starts from the first element in the pool and calls the callback for each element until it finds the first free entry. It then replaces the `Terminal` value with a new `Terminal` value with the additional callback. From that point on the `append` method is responsible for scheduling that callback for subsequently added elements. Note that all three operations call `expand` to add an additional block once the current block is empty, to ensure lock-freedom.

4.1.3 Other Deterministic Dataflow Abstractions

There exist other `FlowPool` implementations besides the one just shown. Multi-lane FlowPools [Schlatter et al.(2012)Schlatter, Prokopec, Miller, Haller, and Odersky] have better horizontal scalability and target applications where there are multiple producers. This implementation retains the lock-freedom and linearizability semantics, with the advent of increased scalability due to fewer points of contention. Compared to the previously shown `FlowPool` implementation they have no ordering guarantees even if there is only a single thread at a time.

Single-assignment streams that have well-defined ordering guarantees date back to Oz [Roy and Haridi(2004)]. In Oz they are implemented with single-assignment variables, and they can be implemented in a similar manner using `Future`s and `Promise`s.

`FlowSeq`s [Schlatter et al.(2013)Schlatter, Prokopec, Miller, Haller, and Odersky] are a concurrent, single-assignment sequence abstraction in which values have a well-defined ordering and can be accessed by an index. The `FlowSeq` programming model is also deterministic, and operations are non-blocking and linearizable. `FlowSeq`s can be used in Scala `for`-comprehensions like `Future`s and `FlowPool`s.

Reactive values and containers [Prokopec et al.(2014a)Prokopec, Haller, and Odersky] model dataflow using first-class event streams, but they are limited to a single thread, called an isolate. To achieve concurrency, multiple isolates can exchange events through entities called channels, but using isolates and channels results in non-deterministic programs.

4.2 Snapshot-Based Parallelization

Unlike the previous section, in this section we once more assume that the elements are present in the data structure when the operation starts. While the quiescence assumption had to be implicitly dropped in the last section, in this section we drop this assumption intentionally. We focus on data structures that can be concurrently modified without restrictions – for such data structures it is typically hard for the programmer to guarantee quiescence.

Futures can be modified at a single location and only once. Concurrent queues underlying flow pools shown in Section 4.1.2 can only be modified at the tail, but can be adapted to be modified at the head as well. Data structure like concurrent maps and sets, concurrent linked lists and concurrect priority queues are different in this regard. They allow operations such as the atomic lookup, insert and remove, which can occur anywhere in the data structure. Parallelizing bulk operations on such data structures is considerably more difficult, since the threads participating in the operation have to agree on whether their modifications occur before or after the point when the bulk operation occurs.

Traditionally, operations that require global data structure information or induce a global change in the data structure, such as size retrieval, iteration or removing all the elements, are implemented so that atomicity is guaranteed only during a quiescent period (i.e. a period of time during which no other concurrent operations is in progress), or require some sort of a global lock to eventually ensure quiescence. Java `ConcurrentHashMap`, the

`ConcurrentSkipListMap` and the `ConcurrentLinkedQueue` [Lea(2014)] from the JDK are such examples.

The goal of this section is to show that certain concurrent data structures support an efficient, lock-free, linearizable, lazy snapshot operation. Data-parallel and other bulk operations on these data structures are built from the atomic snapshot, and guarantee linearizability as a consequence. The snapshot is applied to a scalable concurrent map implementation called a Ctrie, and we show that this extension is accompanied with a minimal performance penalty, both in terms of running time and memory usage.

Interestingly, applying lazyness to concurrent data structures allows parallel copying, and improves the scalability of the snapshot operation. This is particularly evident for tree-like data structures like the Ctrie, which is shown in this section. While adding lazyness to persistent data structures clears the road to amortization [Okasaki(1998)], adding lazyness to concurrent data structures is the path to *scalable parallel snapshots*.

The task of providing linearizable snapshot semantics seems to coincide with that of (still unavailable) hardware transactional memories [Knight(1986)] [Herlihy and Moss(1993)], and software transactional memories [Shavit and Touitou(1995)], whose task is to automatically provide transactional semantics to arbitrary operations, as part of the program runtime. Despite their obvious convenience, and the cleverness in many STM implementations, STMs have still not caught on. A part of the reason for this is that most STMs incur relatively large performance penalties for their generality. Compared to the Ctrie snapshot operation, an atomic snapshot on a transactional map collection is much slower. The reason for this is that the atomic snapshots are specifically designed for concurrent data structures such as the Ctrie, and do not have to pay the price of STMs' generality.

Before showing how concurrent, atomic, lock-free snapshots are implemented for the Ctrie concurrent map data structure, we will illustrate the idea of the atomic snapshot operation on a simpler concurrent data structure, namely, the concurrent linked list.

4.2.1 Concurrent Linked Lists

A lock-free concurrent linked list [Harris(2001)] is a linked list implementation that supports thread-safe, linearizable insert and delete operations. In this section we will assume that the linked list holds an ordered set of elements. We note that the implementation due to Harris is not directly applicable to most managed runtimes, as reference marking is not possible there. Rather than working around this obstacle, we limit the linked list operation in this section to insertion, for simplicity.

```
class ConcurrentLinkedList[T] {
  class INode(val elem: T, atomic var main: INode)

  private val end = new INode(posInf, null)
  private atomic var root = new INode(negInf, end)

  private def insert[T](n: INode, elem: T) {
    val main = READ(n.main)
    if (main == end || main.elem >= elem) {
      val nnode = new INode(elem, main)
      if (!CAS(n.main, main, nnode)) insert(n, elem)
    } else insert(main)
  }

  def insert[T](elem: T) {
    insert(READ(root), elem)
  }
}
```

Figure 4.9: Lock-Free Concurrent Linked List

Concurrent Insertions

We show an implementation of a concurrent linked list that contains a sorted set of elements and supports the `insert` operation in Figure 4.9. This linked list internally maintains a list of nodes we will call *I-nodes*. The `insert` operation takes $O(n)$ time, where n is the number of elements in the list at the linearization point. This implementation maintains two references to the nodes at the root and the end of the linked list. These sentinel nodes contain the values `negInf` and `posInf` denoting the element smaller and bigger than all the other elements, respectively. The `insert` method traverses the nodes of the linked list until it finds a node such that its next node contains an element greater than the element that needs to be inserted. We will call the pointer to the next node `main`.

The correctness of the `insert` crucially depends on three properties. First, the elements of the list are always sorted. Second, once added to the list, the node is never removed. Third, the element in the root node is smaller than any other element. These properties inductively ensure that once the private `insert` method is called with some node `n` as an argument, all the nodes preceding `n` will forever contain elements smaller than `n.elem`. In the same time, the `insert` method ensures that the argument `elem` is always greater than `n.elem`. It follows that the location to insert `elem` must always be after `n.elem`. The private `insert` method simply checks the local condition that `n.main.elem` is additionally greater than `elem`, meaning that `elem` must be between `n.elem` and `n.main.elem`. It then attempts to atomically insert `elem` with a CAS instruction.

It remains to be seen if traversing this concurrent data structure can cause consistency problems. In other words, is a snapshot really necessary to atomically traverse this data structure? It turns out that the answer is yes, as illustrated by the following scenario. Assume that the linked list is traversed by a thread T2 when some elements B and

```
def GCAS(in, old, n) = atomic {
  r = READ(in.main)
  if r = old ∧ in.gen = READ(root).gen {
    WRITE(in.main, n)
    return ⊤
  } else return ⊥
}
```

Figure 4.10: GCAS Semantics

D are added to the list $\{A, C, E\}$ in that sequence by a thread T1. Let the mutual ordering of elements be $A < B < C < D < E$. The thread T2 that traverses the linked list concurrently can traverse A and C, be preempted, and traverse D and E after T1 completes its insertions. The result is that T2 observes list state $\{A, C, D, E\}$, while the intermediate list states were $\{A, C, E\}$, $\{A, B, C, E\}$ and $\{A, B, C, D, E\}$.

Such a consistency violation is detrimental to understanding the semantics of a parallel program. We next study how this simple data structure can be augmented with a snapshot operation. The key to doing so will be to invalidate any writes that occur at the moment the snapshot is being taken.

GCAS Procedure

We will modify the `INode` class to hold an additional field called `gen`. This field will hold instances of a class `Gen`. There will be one such `Gen` instance created for each snapshot – it will serve as a unique generation tag. Next, we add a field `prev` with the type `AnyRef`. Finally, we define a method `isMainNode` that checks if a node is an instance of the I-node type:

```
class INode(val gen: Gen, val elem: T, atomic var main: INode) {
  atomic var prev: AnyRef = null
}

def isMainNode(n: AnyRef) = n.isInstanceOf[INode[_]]
```

The `GCAS` procedure has the following preconditions. It takes 3 parameters – an I-node `in`, and two main nodes `old` and `n`. Only the thread invoking the `GCAS` procedure may have a reference to the main node `n`[4]. Each I-node is a mutable placeholder that points to some main node[5]. Each I-node must contain an additional immutable field `gen`. The `in.main` field is only read using the `GCAS_READ` procedure. Each main node must contain an additional field `prev` that is not accessed by the clients.

The GCAS semantics are equivalent to an atomic block shown in Figure 4.10. The `GCAS`

[4] This is easy to ensure in environments with automatic memory management and garbage collection. Otherwise, a technique similar to the one proposed by Herlihy [Herlihy(1993)] can be used to ensure that some other thread does not reuse objects that have already been recycled.

[5] In the case of a linked list data structure the main node is also the I-node.

```
129 def GCAS(in, old, n)
130   WRITE(n.prev, old)
131   if CAS(in.main, old, n) {
132     GCAS_Commit(in, n)
133     return READ(n.prev) = null
134   } else return ⊥
135
136 def GCAS_Commit(in, m)
137   p = READ(m.prev)
138   r = ABORTABLE_READ(root)
139   p match {
140     case n if isMainNode(n) =>
141       if (r.gen = in.gen ∧ ¬readonly) {
142         if CAS(m.prev, p, null) return m
143         else return GCAS_Commit(in, m)
144       } else {
145         CAS(m.prev, p, new Failed(p))
146         return GCAS_Commit(in, READ(in.main))
147       }
148     case fn: Failed =>
149       if CAS(in.main, m, fn.prev) return fn.prev
150       else return GCAS_Commit(in, READ(in.main))
151     case null => return m
152   }
153
154 def GCAS_READ(in)
155   m = READ(in.main)
156   if (READ(m.prev) = null) return m
157   else return GCAS_Commit(in, m)
```

Figure 4.11: GCAS Operations

is similar to a CAS instruction with the difference that it also compares if the I-node `gen` field is equal to the `gen` field of the data structure root. The `GCAS` instruction is also lock-free. We show the implementation in Figure 4.11, based on single-word CAS instructions. The idea is to communicate the intent of replacing the value in the I-node and check the generation field in the root before committing to the new value.

The `GCAS` procedure starts by setting the `prev` field in the new main node `n` to point at main node `old`, which will be the expected value for the first CAS. Since the preconditions state that no other thread sees `n` at this point, this write is safe. The thread proceeds by proposing the new value `n` with a CAS instruction in line 131. If the CAS fails then `GCAS` returns ⊥ and the CAS is the linearization point. If the CAS succeeds (shown in Figure 4.12B), the new main node is not yet committed – the generation of the root has to be compared against the generation of the I-node before committing the value, so the tail-recursive procedure `GCAS_Commit` is called with the parameter `m` set to the proposed value `n`. This procedure reads the previous value of the proposed node and the data structure `root` (we explain the `ABORTABLE_READ` procedure shortly – for now we can assume it is a normal atomic `READ`). It then inspects the previous value.

If the previous value is a main node different than `null`, the root generation is compared to the I-node generation. If the generations are equal, the `prev` field in `m` must be set to `null` to complete the `GCAS` (Figure 4.12C). If the CAS in the line 142 fails, the procedure

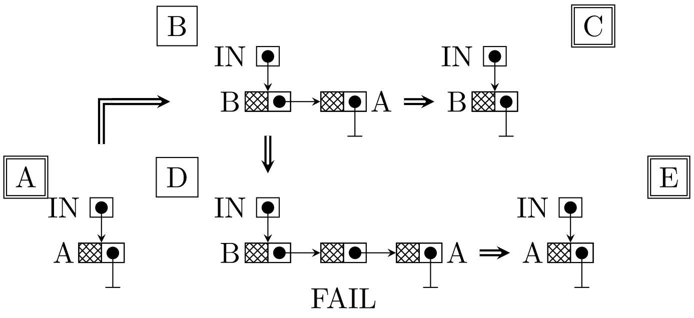


Figure 4.12: GCAS States

is repeated. If the generations are not equal, the `prev` field is set to a special `Failed` node whose previous value is set to `m.prev` (Figure 4.12D), and the `GCAS_Commit` procedure is repeated. This special node signals that the `GCAS` has failed and that the I-node main node must be set back to the previous value.

If the previous value is a failed node, then the main node of the I-node is set back to the previous value from the failed node by the CAS in the line 149 (Figure 4.12D,E). If the CAS is not successful, the procedure must be repeated after rereading the main node.

If the previous value is `null`, then some other thread already checked the generation of the root and committed the node, so the method just returns the current node.

Once the `GCAS_Commit` procedure returns, `GCAS` checks if the `prev` field of the proposed node is `null`, meaning that the value had been successfully committed at some point.

If the proposed value is rejected, the linearization point is the CAS in line 149, which sets the main node of an I-node back to the previous value (this need not necessarily be done by the current thread). If the proposed value is accepted, the linearization point is the successful CAS in the line 142 – independent of that CAS was done by the current thread or some other thread. If the linearization point is external [Bronson(2011a)], we know it happened after `GCAS` was invoked. We know that the `gen` field does not change during the lifetime of an I-node, so it remains the same until a successful CAS in the line 142. If some other thread replaces the root with a new I-node with a different `gen` field after the read in the line 138, then no other thread that observed the root change will succeed in writing a failed node, since we assumed that the CAS in the line 142 succeeded.

To ensure lock-freedom, the `GCAS_READ` procedure must help commit if it finds a proposed value. After reading the main node, it checks if its `prev` field is set to `null`. If it is, it can safely return the node read in line 155 (which is the linearization point) since the algorithm never changes the `prev` field of a committed node and comitting a node sets the `prev` field to `null`. If `prev` is different than `null`, the node hasn't been committed yet, so `GCAS_Commit` is called to complete the read. In this case, the value returned

```
def RDCSS(ov, ovmain, nv)
  r = READ(root)
  if r = ov ∧ GCAS_READ(ov.main) = ovmain {
    WRITE(root, nv)
    return ⊤
  } else return ⊥
```

Figure 4.13: Modified RDCSS Semantics

by GCAS_Commit is the result and the linearization points are the same as with GCAS invoking the GCAS_Commit.

Both GCAS and GCAS_READ are designed to add a non-significant amount of overhead compared a single CAS instruction and a read, respectively. In particular, if there are no concurrent modifications, a GCAS_READ amounts to an atomic read of the node, an atomic read of its prev field, a comparison and a branch.

Applying GCAS

How to use the GCAS procedure to allow linearizable snapshots on a concurrent linked list? First, we augment the internal data structure according to the GCAS preconditions – we identify I-nodes and main nodes, and add the gen and prev fields to these nodes, respectively. Then, we replace all occurrences of CAS instructions with GCAS invocations and we replace all the atomic READ operations with GCAS_READ invocations, except the read of root.

GCAS is only one of the requirements for linearizable snapshots on lock-free data structures. If we only did the GCAS-modifications, the operations would detect a snapshot, but would then restart repetitively. The second requirement is augmenting the operations to rebuild the data structure when they detect it is necessary – in essence, to do a *copy-on-snapshot*.

We show the complete lock-free concurrent linked list with linearizable snapshots implementation in Figure 4.14. The snapshot operations must atomically change the root reference of the linked list to point to an I-node with a fresh generation tag. The CAS instruction alone is insufficient for this task. The snapshot operation can copy and replace the root I-node only if its main node does not change between the copy and the replacement. If it changed, an insert near the root could potentially be lost. This is why we need a stronger primitive that writes only if there are changes to the main node pointed to the root and the root reference itself.

We will use the RDCSS procedure [Harris et al.(2002)Harris, Fraser, and Pratt] to ensure that the snapshot only succeeds if the root node did not change. This (here, software-based) variant of the CAS instruction allows the write to succeed if and only if both the written memory location and a separate memory location contain the expected values.

```
class ConcurrentSnapshotLinkedList[T] {
  class Gen
  class INode(val gen: Gen, val elem: T, atomic var main: INode) {
    atomic var prev: AnyRef = null
  }

  private val end = new INode(gen, posInf, null)
  private atomic var root = new INode(gen, negInf, end)

  private def insert[T](g: Gen, n: INode, elem: T): Boolean = {
    val main = GCAS_READ(n.main)
    if (main.gen == g) {
      if (main == end || main.elem >= elem) {
        val nmain = new INode(elem, main)
        if (!GCAS(n, main, nmain)) false
      } else insert(main)
    } else {
      val nmain = new INode(g, main.elem, main.main)
      if (GCAS(n, main, nmain)) insert(g, n, elem)
      else false
    }
  }

  def insert[T](elem: T) {
    val r = RDCSS_READ(root)
    if (!insert(r.gen, r, elem)) insert(elem)
  }

  def snapshot(): ConcurrentSnapshotLinkedList[T] = {
    val r = RDCSS_READ(root)
    val rmain = GCAS_READ(r.main)
    if (RDCSS(r, rmain, new INode(new Gen, negInf, rmain))) {
      val csl = new ConcurrentSnapshotLinkedList
      csl.root = new INode(new Gen, negInf, rmain)
      csl
    } else snapshot()
  }
}
```

Figure 4.14: Lock-Free Concurrent Linked List with Linearizable Snapshots

It is, in effect, a CAS instruction with two conditions. RDCSS works in a similar way as GCAS, but proposes the new value by creating an intermediate descriptor object, which points to the previous and the proposed value. The allocation cost that we initially wanted to avoid is not critical here, since we expect a snapshot to occur much less often than the other update operations. We specialize RDCSS – the first compare is on the root and the second compare is always on the main node of the old value of the root. GCAS_READ is used to read the main node of the old value of the root. The semantics correspond to the atomic block shown in Figure 4.13.

Recall now the ABORTABLE_READ in line 138 in Figure 4.11. The ABORTABLE_READ is a of RDCSS_READ that writes back the old value to the root field if it finds the proposal descriptor there, causing the snapshot to be restarted. Without the ABORTABLE_READ, two threads that simultaneously start a GCAS on the root I-node and an RDCSS on the root field of the linked list would wait on each other forever.

With GCAS the actual `snapshot` operation becomes a simple $O(1)$ operation. Importantly, note that the modified `insert` operation does not need to rebuild the parts of the concurrent linked list data structure that it does not touch. While this is a constant-factor optimization that does not change the inherent $O(n)$ operation complexity of inserting to a linked list, it opens an interesting optimization opportunity that we will explore in the next section.

4.2.2 Ctries – Concurrent Hash Tries

In this section we show how to apply a snapshot-based parallelization to a more complex data-structure called a Ctrie [Prokopec et al.(2011b)Prokopec, Bagwell, and Odersky] [Prokopec et al.(2012a)Prokopec, Bronson, Bagwell, and Odersky]. A Ctrie can be used to implement efficient, concurrent, lock-free maps and sets. We start by describing the Ctrie data structure and its basic operations, and then augment the Ctrie to support a constant time, linearizable, lock-free snapshot operation. Finally, we show how to use the snapshot operation to implement various atomic parallel operations.

Motivation

Many algorithms exhibit an interesting interplay of parallel traversal and concurrent updates. One such example is the PageRank algorithm, implemented using Scala parallel collections [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky] in Figure 4.15. In a nutshell, this iterative algorithm updates the rank of the pages until the rank converges. The rank is updated based on the last known rank of the pages linking to the current page (line 4). Once the rank becomes smaller than some predefined constant, the page is removed from the set of pages being processed (line 5). The *for* loop that does the updates is executed in parallel. After the loop completes, the arrays containing the previous and the next rank are swapped in line 7, and the next iteration commences if there are pages left.

The main point about this algorithm is that the set of pages being iterated is updated by the remove operation during the parallel traversal. This is where most concurrent data structures prove inadequate for implementing this kind of algorithms – an iterator may or may not reflect the concurrent updates. Scala parallel collections can remedy this by removing the test in line 5 and adding another parallel operation `filter` to create a new set of pages without those that converged – this new set is traversed in the next iteration. The downside of this is that if only a few pages converge during an iteration then almost the entire set needs to be copied. The `filter` approach is not applicable to inputs with short convergence tails. Alternatively, we could implement PageRank by always traversing the same list of pages, and call `setMembership` to disable boolean flags of pages that have already converged. This way we do not need to rebuild the dataset in each iteration. The downside of this approach is that we need to traverse all the pages at the end of the algorithm, when there are only a few pages left. We say that the `setMembership` approach is not applicable to long-convergence tails. If the splitters used for parallel traversal reflected only the elements present when the operation began, both of these issues would be addressed.

```
while (pages.nonEmpty) {
  for (page <- pages.par) {
    val sum = page.incoming.sumBy(p => last(p) / p.links)
    next(page) = (1 - damp) / N + damp * sum
    if (next(page) - last(page) < eps) pages.remove(page)
  }
  swap(next, last)
}
```

Figure 4.15: Parallel PageRank Implementation

Basic operations

Hash array mapped tries [Bagwell(2001)] [Baskins(2000)] (or simply, hash tries) are trees composed of internal nodes and leaves. Leaves store key-value bindings. Internal nodes have a 2^W-way branching factor. In a straightforward implementation, each internal node is a 2^W-element array. Finding a key proceeds as follows. If the internal node is at the level l, then the W bits of the hashcode starting from the position $W * l$ are used as an index to the appropriate branch in the array. This is repeated until a leaf or an empty entry is found. Insertion uses the key to find an empty entry or a leaf. It creates a new leaf with the key if an empty entry is found. Otherwise, the key in the leaf is compared against the key being inserted. If they are equal, the existing leaf is replaced with a new one. If they are not equal (meaning their hashcode prefixes are the same) then the hash trie is extended with a new level.

A more space-efficient version of HAMT was worked on independently by Bagwell [Bagwell(2001)] and Baskins [Baskins(2000)]. Each internal node contains a bitmap of length 2^W. If a bit is set, then the corresponding array entry contains either a branch or a leaf. The array length is equal to the number of bits in the bitmap. The corresponding array index for a bit on position i in the bitmap bmp is calculated as $\#((i-1) \odot bmp)$, where $\#(\cdot)$ is the bitcount and $\odot$ is a bitwise AND operation. The W bits of the hashcode relevant at some level l are used to compute the bit position i as before. At all times an invariant is preserved that the bitmap bitcount is equal to the array length. Typically, W is 5 since that ensures that 32-bit integers can be used as bitmaps. Figure 4.16A shows a hash trie example.

The goal is to create a concurrent data structure that preserves the space-efficiency of hash tries and the expected depth of $O(log_{2^W}(n))$. Lookup, insert and remove will be based solely on CAS instructions and have the lock-freedom property. Remove operations must ensure that the trie is kept as compact as possible. Finally, to support linearizable lock-free iteration and size retrievals, the data structure must support an efficient snapshot

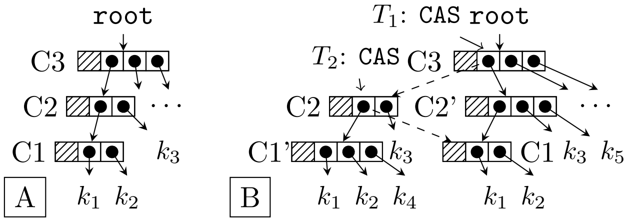


Figure 4.16: Hash Tries

operation. We will call this data structure a *Ctrie*.

Intuitively, a concurrent insertion operation could start by locating the internal node it needs to modify and then create a copy of that node with both the bitmap and the array updated with a reference to the key being inserted. A reference to the newly created node could then be written into the array of the parent node using the CAS instruction. Unfortunately, this approach does not work. The fundamental problem here is due to races between an insertion of a key into some node $C1$ and an insertion of another key into its parent node $C2$. One scenario where such a race happens is shown in Figure 4.16. Assume we have a hash trie from the Figure 4.16A and that a thread T_1 decides to insert a key k_5 into the node $C2$ and creates an updated version of $C2$ called $C2'$. It must then do a CAS on the first entry in the internal node $C3$ with the expected value $C2$ and the new value $C2'$. Assume that another thread T_2 decides to insert a key k_4 into the node $C1$ before this CAS. It will create an updated version of $C1$ called $C1'$ and then do a CAS on the first entry of the array of $C2$ – the updated node $C1'$ will not be reachable from the updated node $C2'$. After both threads complete their CAS operations, the trie will correspond to the one shown in Figure 4.16B, where the dashed arrows represent the state of the branches before the CASes. The key k_4 inserted by the thread T_2 is lost.

We solve this problem by introducing indirection nodes, or I-nodes, which remain present in the Ctrie even as nodes above and below change. The CAS instruction is performed on the I-node instead of on the internal node array. We show that this eliminates the race between insertions on different levels.

The second fundamental problem has to do with the remove operations. Insert operations extend the Ctrie with additional levels. A sequence of remove operations may eliminate the need for the additional levels – ideally, we would like to keep the trie as compact as possible so that the subsequent lookup operations are faster. In Section 4.2.2 we show that removing an I-node that appears to be no longer needed may result in lost updates. We describe how to remove the keys while ensuring compression and no lost updates.

The Ctrie data structure is described in Figure 4.17. Each Ctrie contains a root reference to a so-called indirection node (I-node). An I-node contains a reference to a single node called a *main node*. There are several types of main nodes. A tomb node (T-node) is a

```
structure Ctrie {
  root: INode
  readonly: boolean
}

structure Gen

structure INode {
  main: MainNode
  gen: Gen
}

MainNode:
  CNode | TNode | LNode

Branch:
  INode | SNode
```

```
structure CNode {
  bmp: integer
  array: Branch[2^W]
}

structure SNode {
  k: KeyType
  v: ValueType
}

structure TNode {
  sn: SNode
}

structure LNode {
  sn: SNode
  next: LNode
}
```

Figure 4.17: Ctrie Data Types

special node used to ensure proper ordering during removals. A list node (L-node) is a leaf node used to handle hash code collisions by keeping such keys in a list. These are not immediately important, so we postpone discussion about T-nodes and L-nodes until Sections 4.2.2 and 4.2.2, respectively. A Ctrie node (C-node) is an internal main node containing a bitmap and the array with references to *branch nodes*. A branch node is either another I-node or a singleton node (S-node), which contains a single key and a value. S-nodes are leaves in the Ctrie (shown as key-value pairs in the figures).

The pseudocode in Figures 4.18, 4.20, 4.22, 4.23, 4.11, 4.24, 4.25 and 4.26 assumes short-circuiting semantics of the conditions in the *if* statements. We use logical symbols in boolean expressions. Pattern matching constructs match a node against its type and can be replaced with a sequence of *if-then-else* statements – we use pattern matching for conciseness. The colon (`:`) in the pattern matching cases should be read as *has type*. The keyword `def` denotes a procedure definition. Reads, writes and compare-and-set instructions written in capitals are atomic. This high level pseudocode might not be optimal in all cases – the source code contains a more efficient implementation.

Lookup and insert operations

A lookup starts by reading the root and then calls the recursive procedure `ilookup`, which traverses the Ctrie. This procedure either returns a result or a special value `RESTART`, which indicates that the lookup must be repeated.

The `ilookup` procedure reads the main node from the current I-node. If the main node is a C-node, then (as described in Section 4.2.2) the relevant bit `flag` of the bitmap and the index `pos` in the array are computed by the `flagpos` function. If the bitmap does not contain the relevant bit (line 10), then a key with the required hashcode prefix is not present in the trie, so a `NOTFOUND` value is returned. Otherwise, the relevant branch at index `pos` is read from the array. If the branch is an I-node (line 12), the `ilookup` procedure is called recursively at the next level. If the branch is an S-node (line 14), then the key within the S-node is compared with the key being searched – these two keys have the same hashcode prefixes, but they need not be equal. If they are equal, the corresponding value from the S-node is returned and a `NOTFOUND` value otherwise. In all cases, the linearization point is the read in the line 7. This is because no nodes other than I-nodes change the value of their fields after they are created and we know that the main node was reachable in the trie at the time it was read in the line 7 (see Appendix B).

If the main node within an I-node is a T-node (line 17), we try to remove it and convert it to a regular node before restarting the operation. This is described in more detail in Section 4.2.2. The L-node case is described in Section 4.2.2.

```
 1 def lookup(k)
 2   r = READ(root)
 3   res = ilookup(r, k, 0, null)
 4   if res ≠ RESTART return res else return lookup(k)
 5
 6 def ilookup(i, k, lev, parent)
 7   READ(i.main) match {
 8     case cn: CNode =>
 9       flag, pos = flagpos(k.hash, lev, cn.bmp)
10       if cn.bmp ⊙ flag = 0 return NOTFOUND
11       cn.array(pos) match {
12         case sin: INode =>
13           return ilookup(sin, k, lev + W, i)
14         case sn: SNode =>
15           if sn.k = k return sn.v else return NOTFOUND
16       }
17     case tn: TNode =>
18       clean(parent, lev - W)
19       return RESTART
20     case ln: LNode =>
21       return ln.lookup(k)
22   }
```

Figure 4.18: Ctrie Lookup Operation

When a new Ctrie is created, it contains a root I-node with the main node set to an empty C-node, which contains an empty bitmap and a zero-length array (Figure 4.19A). We maintain the invariant that only the root I-node can contain an empty C-node – all other C-nodes in the Ctrie contain at least one entry in their array. Inserting a key k_1 first reads the root and calling the procedure `iinsert`.

The procedure `iinsert` is invoked on the root I-node. This procedure works in a similar way as `ilookup`. If it reads a C-node within the I-node, it computes the relevant bit and the index in the array using the `flagpos` function. If the relevant bit is not in the bitmap (line 31) then a copy of the C-node with the new entry is created using the `inserted` function. The linearization point is a successful CAS in the line 33, which replaces the current C-node with a C-node containing the new key (see Figures 4.19A,B,C where two new keys k_1 and k_2 are inserted in that order starting from an empty Ctrie). An unsuccessful CAS means that some other operation already wrote to this I-node since its

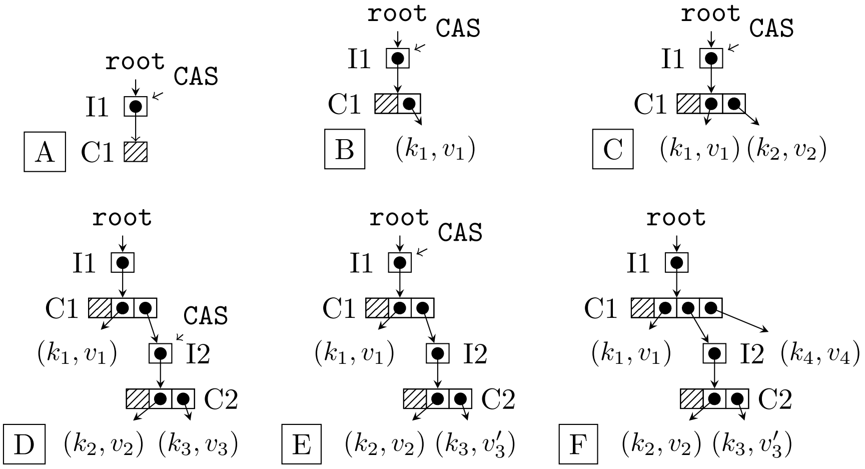


Figure 4.19: Ctrie Insert Illustration

main node was read in the line 28, so the insert must be repeated.

If the relevant bit is present in the bitmap, then its corresponding branch is read from the array. If the branch is an I-node, then `iinsert` is called recursively. If the branch is an S-node and its key is not equal to the key being inserted (line 40), then the Ctrie has to be extended with an additional level. The C-node is replaced with its updated version (line 44), created using the `updated` function that adds a new I-node at the respective position. The new I-node has its main node pointing to a C-node with both keys. This scenario is shown in Figures 4.19C,D where a new key k_3 with the same hashcode prefix as k_2 is inserted. If the key in the S-node is equal to the key being inserted, then the C-node is replaced with its updated version with a new S-node. An example is given in the Figure 4.19E where a new S-node (k_3, v_3') replaces the S-node (k_3, v_3) from the Figure 4.19D. In both cases, the successful CAS instructions in the lines 44 and 48 are the linearization point.

Note that insertions to I-nodes at different levels may proceed concurrently, as shown in Figures 4.19E,F where a new key k_4 is added at the level 0, below the I-node $I1$. No race can occur, since the I-nodes at the lower levels remain referenced by the I-nodes at the upper levels even when new keys are added to the higher levels. This will not be the case after introducing the remove operation.

Remove operation

The remove operation has a similar control flow as the lookup and the insert operation. After examining the root, a recursive procedure `iremove` reads the main node of the I-node and proceeds casewise, removing the S-node from the trie by updating the C-node above it, similar to the insert operation.

```
def insert(k, v)
  r = READ(root)
  if iinsert(r, k, v, 0, null) = RESTART insert(k, v)

def iinsert(i, k, v, lev, parent)
  READ(i.main) match {
    case cn: CNode =>
      flag, pos = flagpos(k.hash, lev, cn.bmp)
      if cn.bmp ⊙ flag = 0 {
        ncn = cn.inserted(pos, flag, SNode(k, v))
        if CAS(i.main, cn, ncn) return OK
        else return RESTART
      }
      cn.array(pos) match {
        case sin: INode =>
          return iinsert(sin, k, v, lev + W, i)
        case sn: SNode =>
          if sn.k ≠ k {
            nsn = SNode(k, v)
            nin = INode(CNode(sn, nsn, lev + W))
            ncn = cn.updated(pos, nin)
            if CAS(i.main, cn, ncn) return OK
            else return RESTART
          } else {
            ncn = cn.updated(pos, SNode(k, v))
            if CAS(i.main, cn, ncn) return OK
            else return RESTART
          }
      }
    case tn: TNode =>
      clean(parent, lev - W)
      return RESTART
    case ln: LNode =>
      if CAS(i.main, ln, ln.inserted(k, v)) return OK
      else return RESTART
  }
```

Figure 4.20: Ctrie Insert Operation

The described approach has certain pitfalls. A remove operation may at one point create a C-node that has a single S-node below it. This is shown in Figure 4.21A, where the key k_2 is removed from the Ctrie. The resulting Ctrie in Figure 4.21B is still valid in the sense that the subsequent insert and lookup operations will work. However, these operations could be faster if (k_3, v_3) were moved into the C-node below $I1$. After having removed the S-node (k_2, v_2), the remove operation could create an updated version of $C1$ with a reference to the S-node (k_3, v_3) instead of $I2$ and write that into $I1$ to compress the Ctrie. But, if a concurrent insert operation were to write to $I2$ just before $I1$ was updated with the compressed version of $C1$, the insertion would be lost.

To solve this problem, we introduce a new type of a main node called a tomb node (T-node). We introduce the following invariant to Ctries – if an I-node points to a T-node at some time t_0 then for all times greater than t_0, the I-node points to the same T-node. In other words, a T-node is the last value assigned to an I-node. This ensures that no inserts occur at an I-node if it is being compressed. An I-node pointing to a T-node is called a *tombed I-node*.

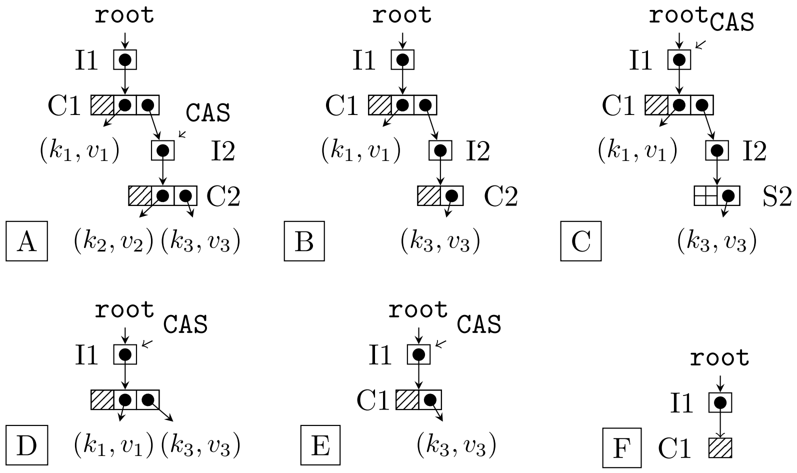


Figure 4.21: Ctrie Remove Illustration

The remove operation starts by reading the root I-node and calling the recursive procedure `iremove`. If the main node is a C-node, the `flagpos` function is used to compute the relevant bit and the branch position. If the bit is not present in the bitmap (line 69), then a `NOTFOUND` value is returned. In this case, the linearization point is the read in the line 66. Otherwise, the branch node is read from the array. If the branch is another I-node, the procedure is called recursively. If the branch is an S-node, its key is compared against the key being removed. If the keys are not equal (line 75), the `NOTFOUND` value is returned and the linearization point is the read in the line 66. If the keys are equal, a copy of the current node without the S-node is created. The *contraction* of the copy is then created using the `toContracted` procedure. A successful CAS in the line 79 will substitute the old C-node with the copied C-node, thus removing the S-node with the given key from the trie – this is the linearization point.

If a given C-node has only a single S-node below and is not at the root level (line 101) then the `toContracted` procedure returns a T-node that wraps the S-node. Otherwise, it just returns the given C-node. This ensures that every I-node except the root points to a C-node with at least one branch. Furthermore, if it points to exactly one branch, then that branch is not an S-node (this scenario is possible if two keys with the same hashcode prefixes are inserted). Calling this procedure ensures that the CAS in the line 79 replaces the C-node $C2$ from the Figure 4.21A with the T-node in Figure 4.21C instead of the C-node $C2$ in Figure 4.21B. This CAS is the linearization point since the S-node (k_2, v_2) is no longer in the trie. However, it does not solve the problem of compressing the Ctrie (we ultimately want to obtain a Ctrie in Figure 4.21D). In fact, given a Ctrie containing two keys with long matching hashcode prefixes, removing one of these keys will create a arbitrarily long chain of C-nodes with a single T-node at the end. We introduced the invariant that no tombed I-node changes its main node. To remove the tombed I-node, the reference to it in the C-node above must be changed with

```
59 def remove(k)
60   r = READ(root)
61   res = iremove(r, k, 0, null)
62   if res ≠ RESTART return res
63   else return remove(k)
64
65 def iremove(i, k, lev, parent)
66   READ(i.main) match {
67     case cn: CNode =>
68       flag, pos = flagpos(k.hash, lev, cn.bmp)
69       if cn.bmp ⊙ flag = 0 return NOTFOUND
70       res = cn.array(pos) match {
71         case sin: INode =>
72           iremove(sin, k, lev + W, i)
73         case sn: SNode =>
74           if sn.k ≠ k
75             NOTFOUND
76           else {
77             ncn = cn.removed(pos, flag)
78             cntr = toContracted(ncn, lev)
79             if CAS(i.main, cn, cntr) sn.v else RESTART
80           }
81       }
82       if res = NOTFOUND ∨ res = RESTART return res
83       if READ(i.main): TNode
84         cleanParent(parent, in, k.hash, lev - W)
85       return res
86     case tn: TNode =>
87       clean(parent, lev - W)
88       return RESTART
89     case ln: LNode =>
90       nln = ln.removed(k)
91       if length(nln) = 1 nln = entomb(nln.sn)
92       if CAS(i.main, ln, nln) return ln.lookup(k)
93       else return RESTART
94   }
```

Figure 4.22: Ctrie Remove Operation

a reference to its *resurrection.* A resurrection of a tombed I-node is the S-node wrapped in its T-node. For all other branch nodes, the resurrection is the node itself.

To ensure compression, the remove operation checks if the current main node is a T-node after removing the key from the Ctrie (line 83). If it is, it calls the `cleanParent` procedure, which reads the main node of the parent I-node `p` and the current I-node `i` in the line 113. It then checks if the T-node below `i` is reachable from `p`. If `i` is no longer reachable, then it returns – some other thread must have already completed the contraction. If it is reachable then it replaces the C-node below `p`, which contains the tombed I-node `i` with a copy updated with the resurrection of `i` (CAS in the line 122). This copy is possibly once more contracted into a T-node at a higher level by the `toContracted` procedure.

To preserve the lock-freedom property, all operations that read a T-node must help compress it instead of waiting for the removing thread to complete the compression. For example, after finding a T-node lookups call the `clean` procedure on the parent node in

```
95  def toCompressed(cn, lev)
96    num = bit#(cn.bmp)
97    ncn = cn.mapped(resurrect(_))
98    return toContracted(ncn, lev)
99
100 def toContracted(cn, lev)
101   if lev > 0 ∧ cn.array.length = 1
102     cn.array(0) match {
103       case sn: SNode => return entomb(sn)
104       case _ => return cn
105     }
106   else return cn
107
108 def clean(i, lev)
109   m = READ(i.main)
110   if m: CNode CAS(i.main, m, toCompressed(m, lev))
111
112 def cleanParent(p, i, hc, lev)
113   m, pm = READ(i.main), READ(p.main)
114   pm match {
115     case cn: CNode =>
116       flag, pos = flagpos(k.hash, lev, cn.bmp)
117       if bmp ⊙ flag = 0 return
118       sub = cn.array(pos)
119       if sub ≠ i return
120       if m: TNode {
121         ncn = cn.updated(pos, resurrect(m))
122         if ¬CAS(pm, cn, toContracted(ncn, lev))
123           cleanParent(p, i, hc, lev)
124       }
125     case _ => return
126   }
```

Figure 4.23: Compression Operations

the line 17. This procedure creates the *compression* of the given C-node – a new C-node with all the tombed I-nodes below resurrected. This new C-node is contracted if possible. The old C-node is then replaced with its compression with the CAS in the line 110. Note that neither `clean` nor `cleanParent` are ever called for the parent of the root, since the root never contains a T-node. For example, removing the S-node (k_3, v_3) from the Ctrie in Figure 4.21D produces a Ctrie in Figure 4.21E. A subsequent remove produces an empty trie in Figure 4.21F.

Both insert and lookup are tail-recursive and may be rewritten to loop-based variants, but this is not so trivial with the remove operation. Since remove operations must be able to compress arbitrary long chains of C-nodes, the call stack is used to store information about the path in the Ctrie being traversed.

The operations shown so far constitute the basic Ctrie operations, i.e. operations working on a single element at a time. We show the correctness, linearizability and lock-freedom proofs [Prokopec et al.(2011a)Prokopec, Bagwell, and Odersky] for the basic Ctrie operations in the Appendix B.

Hash collisions

In this implementation, hash tries use a 32-bit hashcode space. Although hash collisions are rare, it is still possible that two unequal keys with the same hashcodes are inserted. To preserve correctness, we introduce a new type of nodes called list nodes (L-nodes), which are basically persistent linked lists. If two keys with the same hashcodes collide, we place them inside an L-node.

We add another case to the basic operations from Section 4.2.2. Persistent linked list operations `lookup`, `inserted`, `removed` and `length` are trivial and not included in the pseudocode. We additionally check if the updated L-node in the `iremove` procedure has length 1 and replace the old node with a T-node in this case.

Another important change is in the `CNode` constructor in line 42. This constructor was a recursive procedure that creates additional C-nodes as long as the hashcode chunks of the two keys are equal at the given level. We modify it to create an L-node if the level is greater than the length of the hashcode – in our case 32.

Additional operations

Collection classes in various frameworks typically need to implement additional operations. For example, the `ConcurrentMap` interface in Java defines four additional methods: `putIfAbsent`, `replace` any value a key is mapped to with a new value, `replace` a specific value a key is mapped to with a new value and `remove` a key mapped to a specific value. All of these operations can be implemented with trivial modifications to the operations introduced in Section 4.2.2. For example, removing a key mapped to a specific value can be implemented by adding an additional check `sn.v = v` to the line 74.

As argued earlier, methods such as `size`, `iterator` or `clear` commonly seen in collection frameworks cannot be implemented in a lock-free, linearizable manner so easily. The reason for this is that they require global information about the data structure at one specific instance in time – at first glance, this requires locking or weakening the contract so that these methods can only be called during a quiescent state. These methods can be computed efficiently and correctly by relying on a constant time lock-free, atomic snapshot.

Snapshot

While creating a consistent snapshot often seems to require copying all of the elements of a data structure, this is not generally the case. Persistent data structures, present in functional languages, have operations that return their updated versions and avoid copying all the elements, typically achieving logarithmic or sometimes even constant complexity, as we have learned in Chapter 3.

A persistent hash trie data structure seen in standard libraries of languages like Scala or Clojure is updated by rewriting the path from the root of the hash trie to the leaf the key belongs to, leaving the rest of the trie intact. This idea can be applied to implement the snapshot. A generation count can be assigned to each I-node. A snapshot is created by copying the root I-node and setting it to the new generation. When some update operation detects that an I-node being read has a generation older than the generation of the root, it can create a copy of that I-node initialized with the latest generation and update the parent accordingly – the effect of this is that after the snapshot is taken, a path from the root to some leaf is updated only the first time it is accessed, analogous to persistent data structures. The snapshot is thus an $O(1)$ operation, while all other operations preserve an $O(\log n)$ complexity, albeit with a slightly larger constant factor.

Still, the snapshot operation will not work as described above, due to the races between the thread creating the snapshot and threads that have already read the root I-node with the old generation and are traversing the Ctrie in order to update it.The problem is that a CAS that is a linearization point for an insert (e.g. in the line 48) can be preceeded by the snapshot creation – ideally, we want such a CAS instruction to fail, since the generation of the Ctrie root has changed. If we used a DCAS instruction instead, we could ensure that the write occurs only if the Ctrie root generation remained the same. However, most platforms do not support an efficient implementation of this instruction yet. On closer inspection, we find that an RDCSS instruction described by Harris et al. [Harris et al.(2002)Harris, Fraser, and Pratt] that does a double compare and a single swap is enough to implement safe updates. The downside of RDCSS is that its software implementation creates an intermediate descriptor object. While such a construction is general, due to the overhead of allocating and later garbage collecting the descriptor, it is not optimal in our case.

Fortunately, we can use the generation-compare-and-swap, or GCAS, described in the previous section, instead of RDCSS or DCAS. In fact, to compute the Ctrie snapshot, we will use the same approach as we did for concurrent linked lists. Recall that the GCAS has the advantage that it does not create the intermediate object except in the case of failures that occur due to the snapshot being taken – in this case the number of intermediate objects created per snapshot is $O(t)$ where t is the number of threads invoking some modification operation at the time of the snapshot.

Implementation

We now show how to augment the existing algorithm with snapshots using the `GCAS` and `GCAS_READ` procedures. We add a `prev` field to each type of a main node and a `gen` field to I-nodes. The `gen` field points to generation objects allocated on the heap. We do not use integers to avoid overflows and we do not use pointers to the root as generation objects, since that could cause memory leaks – if we did, the Ctrie could potentially transitively point to all of its previous snapshot versions. We add an additional parameter `startgen` to procedures `ilookup`, `iinsert` and `iremove`. This parameter contains the generation count of the Ctrie root, which was read when the operation began.

Next, we replace every occurence of a CAS instruction with a call to the `GCAS` procedure. We replace every atomic read with a call to the `GCAS_READ` procedure. Whenever we read an I-node while traversing the trie (lines 12, 37 and 71) we check if the I-node generation corresponds to `startgen`. If it does, we proceed as before. Otherwise, we create a copy of the current C-node such that all of its I-nodes are copied to the newest generation and use `GCAS` to update the main node before revisiting the current I-node again. This is shown in Figure 4.24, where the `cn` refers to the C-node currently in scope (see Figures 4.18, 4.20 and 4.22). In line 43 we copy the C-node so that all I-nodes directly below it are at the latest generation before updating it. The `readonly` field is used to check if the Ctrie is read-only - we explain this shortly. Finally, we add a check to the `cleanParent` procedure, which aborts if `startgen` is different than the `gen` field of the I-node.

```
...
  case sin: INode =>
    if (startgen eq in.gen)
      return iinsert(sin, k, v, lev + W, i, startgen)
    else
      if (GCAS(cn, atGen(cn, startgen)))
        iinsert(i, k, v, lev, parent, startgen)
      else return RESTART
...
def atGen(n, ngen)
  n match {
    case cn: CNode => cn.mapped(atGen(_, ngen))
    case in: INode => new INode(GCAS_READ(in), ngen)
    case sn: SNode => sn
  }
```

Figure 4.24: I-node Renewal

All `GCAS` invocations fail if the generation of the Ctrie root changes and these failures cause the basic operations to be restarted. Since the root is read once again after restarting, we are guaranteed to restart the basic operation with the updated value of the `startgen` parameter.

As with linked lists, one might be tempted to implement the snapshot operation by simply using a CAS instruction to change the `root` reference of a Ctrie to point to an I-node with a new generation. However, the snapshot operation can copy and replace the root I-node only if its main node does not change between the copy and the replacement.

We again use the `RDCSS` procedure described by Harris to propose the new value by creating an intermediate descriptor object, which points to the previous and the proposed value. Once more, we specialize `RDCSS` to suit our needs – the first compare is on the root and the second compare is always on the main node of the old value of the root.

GCAS_READ is used to read the main node of the old value of the root. The semantics correspond to the atomic block shown in Figure 4.13.

To create a snapshot of the Ctrie the root I-node is read. Its main node is read next. The RDCSS procedure is called, which replaces the old root I-node with its new generation copy. If the RDCSS is successful, a new Ctrie is returned with the copy of the root I-node set to yet another new generation. Otherwise, the snapshot operation is restarted.

```
133 def snapshot()
134   r = RDCSS_READ()
135   expmain = GCAS_READ(r)
136   if RDCSS(r, expmain, new INode(expmain, new Gen))
137     return new Ctrie {
138       root = new INode(expmain, new Gen)
139       readonly = ⊥
140     }
141   else return snapshot()
```

Figure 4.25: Snapshot Operation

An attentive reader will notice that if two threads simultaneously start a GCAS on the root I-node and an RDCSS on the root field of the Ctrie, the algorithm will deadlock[6] since both locations contain the proposed value and read the other location before committing. To avoid this, one of the operations has to have a higher priority. This is the reason for the ABORTABLE_READ in line 138 in Figure 4.11 – it is a modification of the RDCSS_READ that writes back the old value to the root field if it finds the proposal descriptor there, causing the snapshot to be restarted. The algorithm remains lock-free, since the snapshot reads the main node in the root I-node before restarting, thus having to commit the proposed main node.

Since both the original Ctrie and the snapshot have a root with a new generation, both Ctries will have to rebuild paths from the root to the leaf being updated. When computing the size of the Ctrie or iterating the elements, we know that the snapshot will not be modified, so updating paths from the root to the leaf induces an unnecessary overhead. To accomodate this we implement the readOnlySnapshot procedure that returns a read only snapshot. The only difference with respect to the snapshot procedure in Figure 4.25 is that the returned Ctrie has the old root r (line 138) and the readonly field is set to ⊤. The readonly field mentioned earlier in Figures 4.17, 4.11 and 4.24 guarantees that no writes to I-nodes occur if it is set to ⊤. This means that paths from the root to the leaf being read are not rewritten in read-only Ctries. The rule also applies to T-nodes – instead of trying to clean the Ctrie by resurrecting the I-node above the T-node, the lookup in a read-only Ctrie treats the T-node as if it were an S-node. Furthermore, if the GCAS_READ procedure tries to read from an I-node in which a value is proposed, it will abort the write by creating a failed node and then writing the old value back (line 141).

Finally, we show how to implement snapshot-based operations in Figure 4.26. The size

[6] More accurately, it will cause a stack overflow in the current implementation.

```
def iterator()
  if readonly return new Iterator(RDCSS_READ(root))
  else return readOnlySnapshot().iterator()

def size()
  sz = 0
  it = iterator()
  while it.hasNext sz += 1
  return sz

def clear()
  r = RDCSS_READ()
  expmain = GCAS_READ(r)
  if ¬RDCSS(r, expmain, new INode(new Gen)) clear()
```

Figure 4.26: Snapshot-Based Operations

operation can be optimized further by caching the size information in main nodes of a read-only Ctrie – this reduces the amortized complexity of the `size` operation to $O(\log n)$ because the size computation is amortized across the update operations that occurred since the last snapshot. In fact, any associative reduction operation over the elements of the Ctrie can be cached in this way. Furthermore, `size` and other associative operations can be executed in parallel after invoking the `readOnlySnapshot` method, by using the frozen Ctrie as a software combining tree [Herlihy and Shavit(2008)]. We do not go into details nor do we show the entire splitter implementation, which is trivial once a snapshot is obtained.

4.2.3 Snapshot Performance

Having studied the snapshot implementation, we turn to examining several important performance aspects. Unlike the concurrent linked list from Section 4.2.1, which has $O(n)$ asymptotic running time and was introduced as a proof of concept data structure, Ctries have expected $O(\log_{32} n)$ modification and lookup operations, with good scalability and absolute performance (see Section 6.4 for comparison with similar data structures). If extending Ctries with snapshots were to compromise performance, their practicality would be greatly diminished. Fortunately, this is not the case. In this section, we quantify the performance penalties of adding snapshots to Ctries, analyze the worst-case space consumption of a Ctrie augmented with snapshots, compare it the memory footprint of the Ctrie against that of similar data structures, and compare Ctrie snapshot performance to that of using an STM.

Snapshot Overheads

We start by experimentally comparing the absolute performance of the basic Ctrie without snapshot support (*no-snapshot-support*), a Ctrie with snapshot support on which the `snapshot` method was never called (*snapshot-support*), and a Ctrie with snapshot support

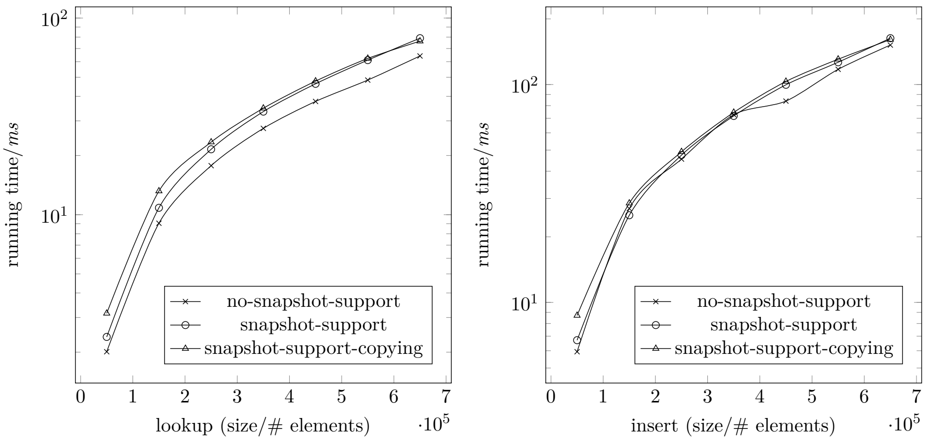


Figure 4.27: Snapshot Overheads with Lookup and Insert Operations

on which the `snapshot` method had been invoked immediately before the benchmark started (*snapshot-support-copying*). We run two simple benchmarks that contrast the costs of replacing CAS with GCAS against the costs of rebuilding the Ctrie after a snapshot. In the first benchmark, we sequentially, i.e. from a single thread, invoke the lookup operation on all the elements contained in the Ctrie. We do so separately for the three Ctrie instances, and plot three separate curves that correlate the number of elements with the running time of the benchmark. The second benchmark evaluates the insert operation, which replaces all the keys in the Ctrie, in a similar way. We run the benchmarks on an Intel i7-4900MQ 3.8 GHz Quad-Core processor with hyperthreading, and compare the running times using a logarithmic y-axis.

These benchmarks show us that augmenting Ctries with support for snapshots increases the running time of lookups by up to 23%. The running time of the insert operation is increased by up to 20%. This indicates that augmenting Ctries with GCAS incurs acceptable overheads.

In both benchmarks, accessing all the elements in the Ctrie after taking a snapshot adds a copying overhead which is 30% for small Ctries, and approaches only 2% as we increase the number of elements. This indicates that the cost of rebuilding the Ctrie is almost completely amortized by the cost of accessing all the elements, particularly for larger Ctries. Note that this result shows the average performance when we access many elements – some initial proportion of accesses, ocurring immediately after taking a snapshot, can require more time, as larger parts of the Ctrie need to be rebuilt.

Importantly, these benchmarks characterize the overheads of extending Ctries with snapshots, and are run on a single thread – they tell us nothing about the scalability of the snapshot operation when the copying proceeds in parallel. More extensive benchmarks, which study the scalability of parallel snapshots, are shown in Section 6.4.

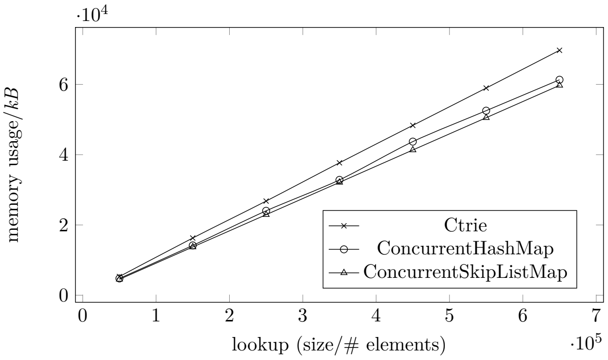


Figure 4.28: Ctries Memory Consumption

Memory Consumption

We now evaluate memory consumption of the Ctrie, both from a theoretical standpoint, with respect to spatial complexity, and empirically, by comparing it to similar data structures.

First, we consider the worst-case space requirements of a Ctrie augmented with snapshots. Note that the `prev` field in each main node points to the previous value at the same I-node. It would be quite unfortunate if a chain of `prev` pointers, captured each time when a snapshot occurs, induced a linked list of main nodes belonging to different generations of the Ctrie, eventually leading to the first occurence of the main node. If this situation were possible, the worst-case space consumption of a Ctrie would be $O(n + t \cdot S)$, where S is the number of snapshots taken during the lifetime of the Ctrie, a potentially unbounded value.

We will prove that this never happens, and that a single Ctrie instance never occupies more than $O(n + t)$ space, where n is the number of elements in the Ctrie, and t is the number of threads executing a modification operation (insert or remove) at the time when the snapshot was taken. To do this, we consider the `GCAS` and `GCAS_READ` methods, and note the following:

- Every main node is created with a non-`null` `prev` field.
- Once the `prev` field is set to `null`, it is never changed again.
- The `GCAS_READ` method at the I-node `in` can only return a main node whose `prev` field is `null`.
- The `GCAS` method at the I-node `in` with the main-node `n` can only return after the `prev` field in `n` is set to `null`.

These lemmas imply that the chain of `prev` nodes referenced by any Ctrie operation can

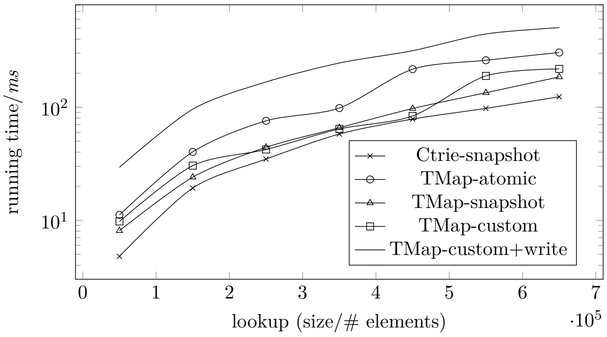


Figure 4.29: Ctrie Snapshots vs STM Snapshots

have the length of at most 2 (due to the `Failed` node). If there are t such operations performed by t separate threads in progress when the snapshot is taken, the total space consumption becomes $O(n + t)$. Admittedly, the number of threads t is unbounded, but this is the case with any lock-free data structure – a system with an unbounded number of threads usually has a separate set of problems.

The theoretical treatment gives us a solid proof of the worst case spatial complexity, but it tells us little about the constant factors involved. In Figure 4.28, we compare the memory footprint of Ctries with snapshots against that of `ConcurrentHashMap` and `ConcurrentSkipListMap` from JDK. We find that our implementation consumes approximately 13% more space than `ConcurrentHashMap` and 17% more space than `ConcurrentSkipListMap`. We consider both these overheads acceptable for typical applications.

Comparison with STMs

Lastly, we contrast the benefits of snapshots against an alternative technology, namely, software transactional memories. As argued earlier, an STM can automatically provide transactional semantics for arbitrary data structures, but pays high runtime overheads for this generality. For this reason, specific STM implementations like ScalaSTM [Bronson et al.(2010b)Bronson, Chafi, and Olukotun] [Bronson(2011b)] provide collections such as transactional arrays and transactional maps as basic primitives. These transactional collections are not implemented directly in terms of transactional references, and are integrated with the STM in a more optimal way. ScalaSTM even goes a step further by providing the transactional snapshot operation on its transactional map collection, and implements it using a data structure similar to the Ctrie.

We compare the performance of the Ctrie snapshot (*Ctrie-snapshot*) against that of ScalaSTM's transactional map snapshot (*TMap-snapshot*), an atomic block that accesses the transactional map (*TMap-atomic*), and a custom transactional hash table, implemented using transactional arrays and transactional references (*TMap-custom*). We also test the custom transactional table by doing just a single insert operation before the lookups, because that forces ScalaSTM to serialize the concurrent transactions, and is reasonable to assume in a real use case (*TMap-custom+write*). The benchmark is similar to an earlier one – in all cases, all the entries in the collection are looked up atomically. This time, however, we use 4 threads that perform the lookups, to simulate the effects of several concurrent transactions.

Figure 4.29 shows that all the other variants are slower than using Ctries. The transactional map snapshot, although specialized, is 50% slower than Ctrie snapshot, and the atomic block that accesses the transactional map is 2.5× slower. The custom transactional hash table, although simple and fast when used from a single thread, is up to 5× slower than Ctrie when at least one write appears in the transaction.

4.2.4 Summary of the Snapshot-Based Parallelization

We applied the snapshot approach to two separate lock-free concurrent data structures – concurrent linked lists and concurrent hash tries. The procedure was similar in both cases:

1. First, we identified the I-nodes and main nodes in the data structure and augmented them with `gen` and `prev` fields.
2. Then, we replaced all occurences of CAS and READ instructions with calls to `GCAS` and `GCAS_READ`, respectively.
3. After that, we modified all the operations that traverse parts of the data structure to pass the generation tag argument. The generation tag is used to update parts of the trie that are out of date, before accessing them.
4. We identified the root of the data structure and replaced all its atomic READ accesses with `RDCSS_READ` calls.
5. Finally, we implemented the snapshot operation using an `RDCSS` call.

Interestingly, the destructive operations on the snapshot of the Ctrie did not change their complexity with respect to the same operations on the non-snapshot Ctrie instance. The copy-on-snapshot does not have to be done eagerly for the entire data structure – instead, it can lazily rebuild only those parts of the data structure traversed by specific operations. Concurrent data structures that have efficient persistent variants seem to be particularly amenable to this technique.

Note that an additional advantage of using lazy snapshots is that the copying can be parallelized if the concurrent data structure is a balanced tree. Once the root of the data structure is renewed, paths from the root to different leafs can be rebuilt independently, so the work of copying is spread over processors subsequently executing concurrent operations.

4.3 Related work

Concurrently accessible queues have been present for a while, an implementation is described by [Mellor-Crummey(1987)]. Non-blocking concurrent linked queues are described by Michael and Scott [Michael and Scott(1996)]. This CAS-based queue implementation is cited and used widely today, a variant of which is present in the Java standard library. More recently, Scherer, Lea and Scott [Scherer et al.(2009)Scherer, Lea, and Scott] describe synchronous queues, which internally hold both data and requests. Both approaches above entail blocking (or spinning) at least on the consumer's part when the queue is empty.

While these concurrent queues fit well in the concurrent imperative model, they have the disadvantage that the programs written using them are inherently nondeterministic. Roy and Haridi [Roy and Haridi(2004)] describe the Oz programming language, a subset of which yields programs deterministic by construction. Oz dataflow streams are built on top of single-assignment variables – using them results in deterministic programs. Here, a deterministic program is guaranteed to, given the same inputs, always yield the same results, or always throw some exception. They allow multiple consumers, but only one producer at a time. Oz has its own runtime which implements blocking using continuations.

The concept of single-assignment variables is embodied in futures proposed by Baker and Hewitt [Henry C. Baker and Hewitt(1977)], and promises first mentioned by Friedman and Wise [Friedman and Wise(1976)]. Futures were first implemented in MultiLISP [Halstead(1985)], and have been employed in many languages and frameworks since. Scala 2.10 futures [Haller et al.(2012)Haller, Prokopec, Miller, Klang, Kuhn, and Jovanovic] define monadic operators and a number of high-level combinators that create new futures. These APIs avoid blocking.

A number of other models and frameworks recognized the need to embed the concept of futures into other data-structures. Single-assignment variables have been generalized to I-Structures [Arvind et al.(1989)Arvind, Nikhil, and Pingali] which are essentially single-assignment arrays. CnC [Burke et al.(2011)Burke, Knobe, Newton, and Sarkar] is a parallel programming model influenced by dynamic dataflow, stream-processing and tuple spaces. In CnC the user provides high-level operations along with the ordering constraints that form a computation dependency graph.

Reactive streams in the Rx framework proposed by Meijer [Meijer(2012)] are an example of an unrestricted reactive computation. Reactive streams are a much more flexible programming model than futures, but they allow creating non-deterministic programs.

Moir and Shavit give an overview of concurrent data structures [Moir and Shavit(2004)]. A lot of research was done on concurrent lists [Harris(2001)], queues and concurrent priority queues. A good introduction to concurrent, lock-free programming is given by Herlihy and Shavit [Herlihy and Shavit(2008)].

While the individual concurrent hash table operations such as insertion or removal can be performed in a lock-free manner as shown by Maged [Michael(2002)], resizing is typically implemented with a global lock. Although the cost of resizing is amortized against operations by one thread, this approach does not guarantee horizontal scalability. Lea developed an extensible hash algorithm that allows concurrent searches during the resizing phase, but not concurrent insertions and removals [Lea(2014)]. Shalev and Shavit give an innovative approach to resizing – split-ordered lists keep a table of hints into a single linked list in a way that does not require rearranging the elements of the linked list when resizing the table [Shalev and Shavit(2006)].

Skip lists store elements in a linked list. There are multiple levels of linked lists that allow logarithmic time insertions, removals and lookups. Skip lists were originally invented by Pugh [Pugh(1990a)]. Pugh proposed concurrent skip lists that achieve synchronization using locks [Pugh(1990b)]. Concurrent non-blocking skip lists were later implemented by Lev, Herlihy, Luchangco and Shavit [Y. Lev and Shavit(2006)] and Lea [Lea(2014)].

Kung and Lehman [Kung and Lehman(1980)] proposed a concurrent binary search tree – their implementation uses a constant number of locks at a time that exclude other insertion and removal operations, while lookups can proceed concurrently. Bronson presents a scalable concurrent AVL tree that requires a fixed number of locks for deletions [Bronson et al.(2010a)Bronson, Casper, Chafi, and Olukotun]. Recently, a non-blocking binary tree was proposed [Ellen et al.(2010)Ellen, Fatourou, Ruppert, and van Breugel].

Tries were proposed by Briandais [De La Briandais(1959)] and Fredkin [Fredkin(1960)]. Trie hashing was applied to accessing files stored on the disk by Litwin [Litwin(1981)]. Litwin, Sagiv and Vidyasankar implemented trie hashing in a concurrent setting, however, they use mutual exclusion locks [Litwin et al.(1989)Litwin, Sagiv, and Vidyasankar]. Hash array mapped trees, or hash tries, are tries for shared-memory described by Bagwell [Bagwell(2001)]. To our knowledge, there is no nonblocking concurrent implementation of hash tries prior to Ctrie.

A persistent data structure is a data structure that preserves its previous version when being modified. Efficient persistent data structures are in use today that re-evaluate only a small part of the data structure on modification, thus typically achieving logarithmic, amortized constant and even constant time bounds for their operations. Okasaki presents an overview of persistent data structures [Okasaki(1998)]. Persistent hash tries have been introduced in standard libraries of languages like Scala and Clojure.

`RDCSS` and `DCAS` software implementations [Harris et al.(2002)Harris, Fraser, and Pratt] are well known. These rely on dynamic memory allocation – this has disadvantages both from the performance and the usability standpoint. Lock-free concurrent deques [Agesen et al.(2000)Agesen, Detlefs, Flood, Garthwaite, Martin, Shavit, and Jr.] that use DCAS were proposed in the past. We note that a wide-spread hardware implementation of these primitives would ease the implementation of many concurrent data structures, including Ctries from this chapter.

4.4 Conclusion

In this chapter, we presented FlowPools for deterministic dataflow computations, referring to their variants that allow increased parallelism. FlowPools have decreased memory consumption compared to classic lock-free queues [Michael and Scott(1996)], and retain similar running time [Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky].

Although determinism is useful when debugging concurrent programs, many applications are in practice inherently nondeterministic. We therefore discourage using FlowPools as a deterministic abstraction, but note that the underlying data structure is an efficient concurrent queue implementation. Concurrent queues serve as buffers between the producers and the consumers, and are inevidable in parallel dataflow frameworks.

In the second part of this chapter, we introduced the Ctrie concurrent data structure. Ctries have particularly scalable insert and remove operations, as we show in Section 6.4. By working towards parallelizing bulk operations on the Ctrie, we introduced an important technique for augmenting concurrent data structures with an efficient, lock-free, linearizable, lazy snapshot operation. Importantly, adding the snapshot operation introduced minimal overheads, both in terms of running time and memory consumption, to existing Ctrie operations.

Lazyness was, for a long time, thought to be useful mainly to provide amortization to persistent data structures, by ensuring that the shared units of work are executed at most once [Okasaki(1996)]. Lazyness in the snapshot ensures that parts of the Ctrie are copied when and if they are needed – multiple processors can simultaneously access and rebuild separate parts of the data structure. In lock-free concurrent data structures, implementing *scalable*, *linearizable*, *lock-free* operations that depend on, or change, the *global* state of the data structure, is a novel use-case for lazy evaluation.

5 Work-stealing Tree Scheduling

Almost every data-parallel workload is to some extent irregular on a multicore or a multiprocessor machine. Even if the number of instructions that need to be executed is equal for every element of the collection, there are many other factors that influence the workload. Some of the workers may wake up slower than other workers or be occupied executing other data-parallel operations, so they can make a workload seem irregular in practice. Then, accessing specific regions of memory might not take the same time for all the processors, and contended writes can slow some processors more than the others, depending on the access pattern. Similarly, a memory allocator might take different amounts of time to return a chunk of memory depending on its internal state. A processor may be preempted by the operating system and unavailable for a certain amount of time, effectively executing its work slower than the rest of the processors. In a managed runtime, a garbage collection cycle or a JIT compiler run can at any time freeze some or all of the worker threads.

Work-stealing [Frigo et al.(1998)Frigo, Leiserson, and Randall] is one solution to load-balancing computations with an irregular workload. In this technique different processors occasionally steal batches of elements from each other to load balance the work – the goal is that no processor stays idle for too long. We have seen an example of task-based data-parallel work-stealing scheduler in Section 2.7. In this chapter we propose and describe a runtime scheduler for data-parallel operations on shared-memory architectures that uses a variant of work-stealing to ensure proper load-balancing. The scheduler relies on a novel data structure with lock-free synchronization operations called the *work-stealing tree.*

5.1 Data-Parallel Workloads

As we saw in previous chapters, data-parallel programs are typically composed from high-level operations, and are declarative rather than imperative in nature. We focus on several concrete data-parallel programs in Figure 5.1. These programs rely heavily on higher-order data-parallel operations such as `map`, `reduce` and `filter`, which take a function argument – they are parametrized by a mapping function, a reduction operator or a filtering predicate, respectively.

The first example in Figure 5.1 computes the variance of a set of measurements `ms`. It starts by computing the mean value using the higher-order operation `sum`, and then `maps` each element of `ms` into a set of squared distances from the mean value, the `sum` of which divided by the number of elements is the variance `v`. The amount of work executed for each measurement value is equal, so we call this workload *uniform*. This need not be always so. The second program computes all the prime numbers from 3 until N by calling a data-parallel `filter` on the corresponding range. The `filter` uses a predicate that checks that no number from 2 to $\sqrt{i}$ divides i. The workload is not uniform nor independent of i and the processors working on the end of the range need to do more work. This example also demonstrates that data-parallelism can be *nested* – the `forall` can be done in parallel as each element may require a lot of work. On the other hand, the `reduce` in the third program that computes a sum of numbers from 0 to N requires a minimum amount of work for each element. A good data-parallel scheduler must be efficient for all the workloads – when executed with a single processor the `reduce` in the third program must have the same running time as the `while` loop in the fourth program, the data-parallelism of which is not immediately obvious due to its imperative style.

```
val sz = ms.size
val a = ms.sum / sz
val ds = ms map {
  x => (x - a)^2
}
val v = ds.sum / sz
```

```
val r = 3 until N
val ps = r filter {
  i =>
  2 to ⌈√i⌉ forall {
    d => i % d != 0
  }
}
```

```
val r = 0 until N
val sum = r reduce {
  (acc, i) =>
    acc + i
}
```

```
var sum = 0
var i = 0
while (i < N) {
  sum += i
  i += 1
}
```

Figure 5.1: Data Parallel Program Examples

As argued in Chapter 2 and illustrated in Figure 5.1, data-parallel operations are highly generic – for example, `reduce` takes a user-provided operator, such as number addition, string concatenation or matrix multiplication. This genericity drives the workload distribution, which cannot always be determined statically. To assign work to processors optimally, scheduling must occur at runtime. Scheduling in this case entails dividing the elements into batches on which the processors work in isolation.

Before assessing the goals and the quality of scheduling, we need to identify different classes of computational workloads. We will refer to workloads that have an equal amount of work assigned to every element as *uniform workload*. A uniform workload that has the least possible possible amount of possible useful work assigned to each element is called the *baseline workload*. When comparing two uniform workloads we say that the workload with less work per element is more *fine-grained*, as opposed to the other that is more *coarse-grained*. All workloads that are not uniform are considered *irregular workloads*.

As argued at the beginning of this section, there is no workload that is truly uniform in practice. Still, the concept of a uniform workload is important when assessing a scheduler and some practical workloads come very close to it.

When evaluating the data-parallel scheduler we first require that the scheduler scales linearly when the workload is the baseline workload. This ensures that the scheduler works well when the workloads are fine-grained. Without this constraint it is trivial to implement a scheduler for irregular workloads – we have already shown one such scheduler in Figure 2.23 in Section 2.9. After ensuring baseline scalability we assess the quality of the scheduler by comparing it against other schedulers on different irregular workloads.

In conclusion, a data-parallel scheduler must meet the following criteria:
C1 There is no noticeable overhead when executing the baseline with a single processor.
C2 Speedup is optimal for both the baseline and for typical irregular workloads appearing in practice.
C3 Speedup is optimal for coarse-grained workloads where the work granularity is on the order of magnitude of the parallelism level.

5.2 Work-stealing Tree Scheduling

In this section we introduce the work-stealing tree scheduling algorithm. We start by introducing the basic work-stealing tree data types and operations, and then introduce the scheduling algorithm workers execute, along with several extensions. For the purposes of this section we will be parallelizing a loop, i.e. a simple range collection that represents a finite interval of integer numbers.

The design of the algorithm is driven by the following assumptions. There are no fast, accurate means to measure elapsed time with sub-microsecond precision, i.e. there is no way to measure the running time of an operation. There is no static or runtime information about the cost of an operation – when invoking a data-parallel operation we do not know how much computation each element requires. There are no hardware-level interrupt handling mechanisms at our disposal – the only way to interrupt a computation is to have the processor check a condition. We assume OS threads as parallelism primitives, with no control over the scheduler. We assume that the available synchronization primitives are monitors and the CAS instruction. CAS can be used for stronger primitives [Harris et al.(2002)Harris, Fraser, and Pratt], but we do not use those directly[1]. We assume the presence of automatic memory management. These constraints are typical for a managed runtime like the JVM.

[1] Using stronger primitives such as software DCAS requires allocating objects, and we avoid it for performance reasons.

5.2.1 Basic Data Types

In this section we describe the work-stealing tree data structure and the scheduling algorithm that the workers run. We first briefly discuss the aforementioned fixed-size batching. We have mentioned that the contention on the centralized queue is one of it drawbacks. We could replace the centralized queue with a queue for each worker and use work-stealing. However, this seems overly eager – we do not want to create as many work queues as there are workers for each parallel operation, as doing so may outweigh the actually useful work. We should start with a single queue and create additional ones on-demand. Furthermore, fixed-size batching seems appropriate for scheduling parallel loops, but what about the reduce operation? If each worker stores its own intermediate results separately, then the `reduce` may not be applicable to non-commutative operators (e.g. string concatenation). It seems reasonable to have the work-stealing data-structure store the intermediate results, since it has the division order information.

With this in mind, we note that a tree seems particularly applicable. When created it consists merely of a single node – a root representing the operation and all the elements of the range. The worker invoking the parallel operation can work on the elements and update its progress by writing to the node it owns. If it completes before any other worker requests work, then the overhead of the operation is merely creating the root. Conversely, if another worker arrives, it can steal some of the work by creating two child nodes, splitting the elements and continuing work on one of them. This proceeds recursively. Scheduling is thus workload-driven – nodes are created only when some worker runs out of work meaning that another worker had too much work. Such a tree can also store intermediate results in the nodes, serving as a reduction tree.

How can such a tree be used for synchronization and load-balancing? We assumed that the parallelism primitives are OS threads. We can keep a pool of threads [Lea(2000)] that are notified when a parallel operations is invoked – we call these workers. We first describe the worker algorithm from a high-level perspective. Each worker starts by calling the tail-recursive `run` method in Figure 5.2. It looks for a node in the tree that is either not already owned or steals a node which some other worker works on by calling `findWork` in line 3. This node is initially a leaf. The worker works on the leaf by calling `workOn` in line 5, which works on it until the leaf until the leaf is either completed or stolen. This is repeated until `findWork` returns $\perp$ (`null`), indicating that all the work is completed.

In Figure 5.2 we also present the *work-stealing tree* and its basic data-types. We use the keyword `struct` to refer to a compound data-type – this can be a Java class or a C structure. We define two compound data-types. `Ptr` is a reference to the tree – it has only a single member `child` of type `Node`. Write access to `child` has to be atomic and globally visible (in Java, this is ensured with the `volatile` keyword). `Node` contains immutable references to the `left` and `right` subtree, initialized upon instantiation. If

```
struct Ptr
  child: Node
struct Node
  left, right: Ptr
  start, until: Int
  progress: Int
  owner: Owner
```

```
def run(): Unit =
  val leaf =
    findWork(root)
  if (leaf ≠ ⊥)
    workOn(leaf)
    run()
```

Figure 5.2: Work-Stealing Tree Data-Types and the Basic Scheduling Algorithm

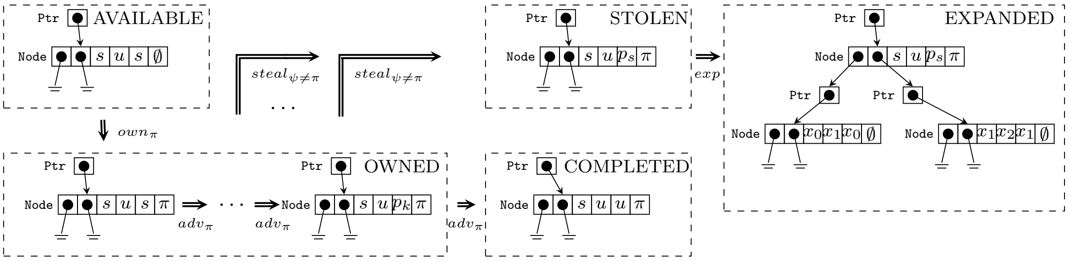


Figure 5.3: Work-Stealing Node State Diagram

these are set to ⊥ we consider the node a leaf. We initially focus on parallelizing loops over ranges, so we encode the current state of iteration with three integers. Members `start` and `until` are immutable and denote the initial range – for the root of the tree this is the entire loop range. Member `progress` has atomic, globally visible write access. It is initially set to `start` and is updated as elements are processed. Finally, the `owner` field denotes the worker that is working on the node. It is initially ⊥ and also has atomic write access. Example trees are shown in Figure 5.3.

Before we describe the operations and the motivation behind these data-types we will define the states work-stealing tree can be in (see Figure 5.3), namely its invariants. This is of particular importance for concurrent data structures which have non-blocking operations. Work-stealing tree operations are lock-free, a well-known advantage [Herlihy and Shavit(2008)], which comes at the cost of little extra complexity in this case.

INV1 Whenever a new node reference `Ptr p` becomes reachable in the tree, it initially points to a *leaf* `Node n`, such that `n.owner = ⊥`. Field `n.progress` is set to `n.start` and `n.until`≥`n.start`. The subtree is in the AVAILABLE state and its range is ⟨`n.start`,`n.until`⟩.

INV2 The set of transitions of `n.owner` is $\bot \to \pi \neq \bot$. No other field of `n` can be written until `n.owner` $\neq \bot$. After this happens, the subtree is in the OWNED state.

INV3 The set of transitions of `n.progress` in the OWNED state is $p_0 \to p_1 \to \ldots \to p_k$ such that `n.start` $= p_0 < p_1 < \ldots < p_k <$ `n.until`. If a worker π writes a value from this set of transitions to `n.progress`, then `n.owner` $= \pi$.

INV4 If the worker `n.owner` writes the value `n.until` to `n.progress`, then that is the

```
 7 def tryOwn(n: Node): Boolean =
 8   if (READ(n.owner) ≠ ⊥) false
 9   else if (CAS(n.owner, ⊥, π)) true
10   else tryOwn(n)
11
12 def tryAdvance(n: Node, p: Int): Int =
13   val q = min(p + STEP, n.until)
14   if (¬CAS(n.progress, p, q)) -1
15   else q - p
16
17 def isLeaf(n: Node): Boolean =
18   n.left == ⊥
19
20 def isEligible(n: Node): Boolean =
21   n.until - READ(n.progress) > 1
```

```
22 def trySteal(ptr: Ptr): Boolean =
23   val c_t0 = READ(ptr.child)
24   if (¬isLeaf(c_t0)) true else
25     val p_t1 = READ(c_t0.progress)
26     if (p_t1 == c_t0.until) false
27     else if (p_t1 ≥ 0)
28       val negp = -p_t1 - 1
29       CAS(c_t0.progress, p_t1, negp)
30       trySteal(ptr)
31     else
32       val c_exp = newExpanded(c_t0)
33       if (CAS(ptr.child, c_t0, c_exp))
34         true
35       else trySteal(ptr)
```

Figure 5.4: Basic Work-Stealing Tree Operations

last transition of `n.progress`. The subtree goes into the COMPLETED state.

INV5 If a worker ψ overwrites p_i, such that $\texttt{n.start} \leq p_i < \texttt{n.until}$, with $p_s = -p_i - 1$, then $\psi \neq \texttt{n.owner}$. This is the last transition of `n.progress` and the subtree goes into the STOLEN state.

INV6 The field `p.child` can be overwritten only in the STOLEN state, in which case its transition is $\texttt{n} \rightarrow \texttt{m}$, where `m` is a copy of `n` with `m.left` and `m.right` being fresh leaves in the AVAILABLE state with ranges $r_l = \langle x_0, x_1 \rangle$ and $r_r = \langle x_1, x_2 \rangle$ such that $r_l \cup r_r = \langle p_i, \texttt{n.until} \rangle$. The subtree goes into the EXPANDED state.

This seemingly complicated set of invariants can be summarized in a straightforward way. Upon owning a leaf, that worker processes elements from that leaf's range by incrementing the `progress` field until either it processes all elements or another worker requests some work by invalidating `progress`. If some other worker invalidates `progress` (i.e. steals a node), then the leaf is replaced by a subtree such that the remaining work is divided between the new leaves.

5.2.2 Work-Stealing Tree Operations

So far we have described in very abstract terms how workers execute useful work. As we will see, the `findWork` method will traverse the tree until a node eligible for work is found. At this point the node either becomes owned by the worker or stolen if it is already owned by another worker. Before we show `findWork`, we must introduce some basic work-stealing tree operations.

Now that we have formally defined a valid work-stealing tree data structure, we provide an implementation of the basic operations (Figure 5.4). A worker must attempt to acquire ownership of a node before processing its elements by calling the method `tryOwn`, which returns `true` if the claim is successful. After reading the `owner` field in line 8 and establishing the AVAILABLE state, the worker attempts to atomically push the node into the OWNED state with the CAS in line 9. This CAS can fail either due to a faster worker claiming ownership or spuriously – a retry follows in both cases.

A worker that claimed ownership of a node repetitively calls `tryAdvance`, which attempts to reserve a batch of size `STEP` by atomically incrementing the `progress` field, eventually bringing the node into the COMPLETED state. If `tryAdvance` returns a nonnegative number, the owner is obliged to process that many elements, whereas a negative number is an indication that the node was stolen.

A worker searching for work must call `trySteal` if it finds a node in the OWNED state. This method returns `true` if the node was successfully brought into the EXPANDED state by any worker, or `false` if the node ends up in the COMPLETED state. Method `trySteal` consists of two steps. First, it attempts to push the node into the STOLEN state with the CAS in line 29 after determining that the node read in line 23 is a leaf. This CAS can fail either due to a different steal, a successful `tryAdvance` call or spuriously. Successful CAS in line 29 brings the node into the STOLEN state. Irregardless of success or failure, `trySteal` is then called recursively. In the second step, the expanded version of the node from Figure 5.3 is created by the `newExpanded` method, the pseudocode of which is not shown here since it consists of isolated singlethreaded code. The `child` field in `Ptr` is replaced with the expanded version atomically with the CAS in line 33, bringing the node into the EXPANDED state.

5.2.3 Work-Stealing Tree Scheduling Algorithm

```
36 def workOn(ptr: Ptr): Boolean =
37   val node = READ(ptr.child)
38   var batch = -1
39   do
40     val p = READ(node.progress)
41     if (p >= 0 ∧ p < node.until)
42       batch = tryAdvance(node, p)
43       if (batch ≠ -1)
44         kernel(p, p + batch)
45     else batch = -1
46   while (batch ≠ -1)
47   if (READ(node.progress) ≥ 0)
48     true
49   else
50     trySteal(ptr)
51     false
```

```
52 def findWork(ptr: Ptr): Node =
53   val node = READ(ptr.child)
54   if (isLeaf(node))
55     if (tryOwn(node)) node
56     else if (¬isEligible(node)) ⊥
57     else if (¬trySteal(ptr))
58       findWork(ptr)
59     else
60       val right = node.right
61       if (tryOwn(READ(right.child)))
62         READ(right.child)
63       else findWork(ptr)
64   else
65     val leftsub = findWork(node.left)
66     if (leftsub ≠ ⊥) leftsub
67     else findWork(node.right)
```

Figure 5.5: Finding and Executing Work

We now describe the scheduling algorithm that the workers execute by invoking the `run` method shown in Figure 5.2. There are two basic modes of operation a worker alternates between. First, it calls `findWork`, which returns a node in the AVAILABLE state (line 70). Then, it calls `descend` to work on that node until it is stolen or completed, which calls `workOn` to process the elements. If `workOn` returns `false`, then the node was stolen

```
def run(): Unit =
  val leaf =
    findWork(root)
  if (leaf ≠ ⊥)
    descend(leaf)
    run()
```

```
def descend(leaf: Ptr): Unit =
  val nosteals = workOn(leaf)
  if (¬nosteals)
    val sub = READ(leaf.child).left
    if (tryOwn(READ(sub.child)))
      descend(sub)
```

Figure 5.6: Top-Level Scheduling Algorithm

and the worker tries to descend one of the subtrees rather than searching the entire tree for work. This decreases the total number of `findWork` invocations. The method `workOn` checks if the node is in the OWNED state (line 41), and then attempts to atomically increase `progress` by calling `tryAdvance`. The worker is obliged to process the elements after a successful advance, and does so by calling the `kernel` method, which is nothing more than the `while` loop like the one in Figure 5.1. Generally, `kernel` can be any kind of a workload. Finally, method `findWork` traverses the tree left to right and whenever it finds a leaf node it tries to claim ownership. Otherwise, it attempts to steal it until it finds that it is either COMPLETED or EXPANDED, returning ⊥ or descending deeper, respectively. Nodes with 1 or less elements left are skipped.

We explore alternative `findWork` implementations later. For now, we state the following claim. If the method `findWork` does return ⊥, then all the work in the tree was obtained by different workers that have called `tryAdvance`. This means that all the elements have been processed, except some number of elements $M \cdot C$ distributed across $M < P$ leaf nodes where P is the number of workers and C is the largest value passed to `tryAdvance`. In essence, C is the largest allowed batch size in the algorithm.

In Figure 5.6 we show a slightly more complicated implementation of the worker `run` method. This implementation is purely a performance optimisation. After a worker completes work on a node it can check if the node has been stolen. If the node is stolen, there is a high probability that there is a free node in the left subtree, so the worker attempts to own it before searching the entire tree to find more work.

Note that `workOn` is similar to fixed-size batching. The difference is that an arrival of a worker invalidates the work-stealing tree node, whereas multiple workers simultaneously call `tryAdvance` in fixed-size batching, synchronizing repetitively and causing contention. We will now study the impact this contention has on performance and scalability – we start by focusing on choosing the `STEP` value from Section 5.2.2.

Results of several experiments are shown in the remainder of this section. Experiments were performed on an Intel i7 3.4 GHz quad-core processor with hyperthreading and Oracle JDK 1.7, using the server JVM. The implementation is written in the Scala, which uses the JVM as its backend. JVM programs are typically regarded as less efficient than programs written in a lower language like C. To show that the evaluation is comparative

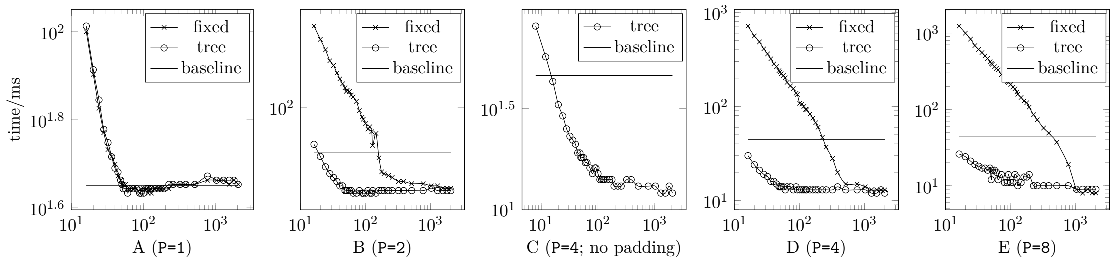


Figure 5.7: Baseline Running Time (ms) vs. STEP Size

to a C implementation, we must confirm that the running time of corresponding sequential C programs is equal to that of the same JVM program. A simple experiment confirms that the running time of the `while` loop from Figure 5.1 is roughly $45ms$ for 150 million elements in both C (GNU C++ 4.2) and on the JVM – if we get linear speedups then we can conclude that the scheduler is indeed optimal. We can thus turn our attention to critera C1 for our data-parallel scheduler.

As hinted at the end of Section 5.1, the `STEP` value in `tryAdvance` from Figure 5.4 should ideally be 1 for load-balancing purposes, but has to be more coarse-grained due to communication costs that could overwhelm the baseline. In Figure 5.7A we plot the running time against the `STEP` size, obtained by executing the baseline loop with a single worker. By finding the minimum `STEP` value with no observable overhead, we seek to satisfy criteria C1. The minimum `STEP` with no noticeable synchronization costs is around 50 elements – decreasing `STEP` to 16 doubles the execution time and for value 1 the execution time is 36 times larger (not shown for readability).

Having shown that the work-stealing tree is as good as fixed-size chunking with a properly chosen `STEP` value, we evaluate its effectiveness with multiple workers. Figure 5.7B shows that the minimum `STEP` for fixed-size chunking increases for 2 workers, as we postulated earlier. Increasing `STEP` decreases the frequency of synchronization and the communication costs associated with it. In this case the 3x slowdown is caused by processors having to exchange ownership of the `progress` field cache-line. The work-stealing tree does not suffer from this problem, since it strives to keep processors isolated – the speedup is linear with 2 workers.

However, with 4 processors the performance of the work-stealing tree is degraded (Figure 5.7C), although for greater `STEP` values the speedup becomes once again linear. The reason for this is not immediately apparent, but it follows from a naive implementation based on the pseudocode in Figure 5.2. The increased communication costs are caused by *false sharing* – even though the two processors work on different nodes, they modify the same cache line, slowing down the CAS in line 14. The reason is that in a JVM implementation memory for the work-stealing nodes is allocated in a contiguous region by one thread that succeeds in expanding work-stealing node after a steal, causing both nodes to end up in the same cache line[2]. Padding the node object with dummy fields to adjust its size to the cache line solves this problem, as shown in Figures 5.7D,E.

Having solved the false-sharing issues, we are still not completely satisfied with the scaling as the number of processors grows. Scalability is dictated by the synchronization costs, which are in turn proportional to the number of created work-stealing nodes. In the next section we focus on decreasing synchronization costs further.

5.2.4 Work-Stealing Node Search Heuristics

Inspecting the number of tree nodes created at different parallelism levels in Figure 5.12B reveals that as the number of workers grows, the number of nodes grows at a superlinear rate. Each node incurs a synchronization cost, so could we decrease their total number?

Examining a particular work-stealing tree instance at the end of the operation reveals that different workers are battling for work in the left subtree until all the elements are depleted, whereas the right subtree remains unowned during this time. As a result, the workers in any subtree steal from each other more often, hence creating more nodes. The cause is the left-to-right tree traversal in `findWork` as defined in Figure 5.5, a particularly bad stealing strategy we will call **Predefined**. As shown in Figure 5.12B, the average tree size for 8 workers nears 2500 nodes. The reason for this is that all the workers consistently try to steal from each other in the left work-stealing subtree, while the right subtree remains occupied by at most a single worker. This pressure on the left subtree causes those nodes to be recursively divided until no more elements remain. Once work in the left subtree is completely consumed, the same sequence of events resumes in the left child of the right subtree. After a certain parallelism level, workers using this strategy start spending more time expanding the work-stealing tree than doing actual useful work.

In this section, we will examine different heuristics for finding work, which lead to better load-balancing. We will refer to these heuristics as *strategies*. Each strategy will redefine the methods `descend` and `findWork`. As we will see, a common trait of all these strategies will be that, if `findWord` returns `null`, then there is no more work to steal in the tree. This is because each strategy will in the worst case visit each leaf of the tree in some order. Note that a leaf that does not contain work at time t_0, does not contain work at any time $t_1 > t_0$, and is a leaf for all $t_1 > t_0$. As a consequence, a stealer that traverses all the nodes, and finds that neither of them contain work, can safely conclude that there is no more work left in the tree.

We first attempt to change the preference of a worker by changing the tree-traversal order in line 64 based on the worker index i and the level l in the tree.

[2] To confirm this, we can inspect the number of elements processed in each work-stealing node. We reveal that the uniform workload is not evenly distributed among the topmost nodes, indicating that the additional time is spent in synchronization.

```
def left(p: Ptr, i: Int) =
  val bit = i >> (p.level % log2(P))
  bit % 2 == 1

def choose(p: Ptr, i: Int) =
  if (left(p, i)) READ(p.child).left
  else READ(p.child).right

def descend(leaf: Ptr): Unit =
  val nosteals = workOn(leaf)
  if (¬nosteals)
    val sub =
      choose(leaf, thisWorker.index)
    if (tryOwn(READ(sub.child)))
      descend(subnode)
```

```
def findWork(ptr: Ptr): Node =
  val node = READ(ptr.child)
  if (isLeaf(node))
    if (tryOwn(node)) node
    else if (¬isEligible(node)) ⊥
    else if (¬trySteal(ptr)) findWork(ptr)
    else
      val ptr =
        choose(ptr, thisWorker.index)
      if (tryOwn(READ(ptr.child)))
        READ(ptr.child)
      else findWork(ptr)
  else if (left(ptr, thisWorker.index))
    val leftsub = findWork(node.left)
    if (leftsub ≠ ⊥) leftsub
    else findWork(node.right)
  else
    val rightsub = findWork(node.right)
    if (rightsub ≠ ⊥) rightsub
    else findWork(node.left)
```

Figure 5.8: **Assign** Strategy

Assign Strategy

In this strategy a worker with index i invoking `findWork` picks a left-to-right traversal order at some node at level l if and only if its bit at position $l \bmod \lceil \log_2 P \rceil$ is 1, that is:

$$(i >> (l \bmod \lceil \log_2 P \rceil)) \bmod 2 = 1 \tag{5.1}$$

This way, the first path from the root to a leaf up to depth $\log_2 P$ is unique for each worker. The consequence of this is that when the workers descend in the tree the first time, they will pick different paths, leading to fewer steals assuming that the workload distribution is relatively uniform. If it is not uniform, then the workload itself should amortize the creation of extra nodes. We give the pseudocode in Figure 5.8.

The choice of the subtree after a steal in lines 77 and 60 is also changed like this. This strategy, which we call **Assign**, decreases the average tree size at $P = 8$ to 134.

AssignTop Strategy

This strategy is similar to the previous one with the difference that the assignment only works as before if the level of the tree is less than or equal to $\lceil log_2 P \rceil$. Otherwise, a random choice is applied in deciding whether traversal should be left-to-right or right-to-left. We show it in Figure 5.9 where we only redefine the method `left`, and reuse the same `choose`, `descend` and `findWork`.

This strategy decreases the average tree size at $P = 8$ to 77.

```
115 def left(p: Ptr, idx: Int) =
116   if (p.level ≤ log2(P))
117     val bit = i >> (p.level % log2(P))
118     bit % 2 == 1
119   else
120     coinToss()
```

```
121 def left(p: Ptr, idx: Int) =
122   coinToss()
123
```

Figure 5.9: **AssignTop** and **RandomAll** Strategies

```
124 def left(p: Ptr, i: Int) =
125   val bit = i >> (p.level % log2(P))
126   bit % 2 == 1
127
128 def choose(p: Ptr, i: Int) =
129   if (left(p, i)) READ(p.child).left
130   else READ(p.child).right
131
132 def descend(leaf: Ptr) =
133   val nosteals = workOn(leaf)
134   if (¬nosteals)
135     val sub = READ(leaf.child).left
136     if (tryOwn(READ(sub.child)))
137       descend(subnode)
```

```
138 def findWork(ptr: Ptr) =
139   val node = READ(ptr.child)
140   if (isLeaf(node))
141     if if (tryOwn(node)) node
142     else if (¬isEligible(node)) ⊥
143     else if (¬trySteal(ptr))
144       findWork(ptr)
145     else
146       val r = node.right
147       if (tryOwn(READ(r.child)))
148         READ(r.child)
149       else findWork(ptr)
150   else if (left(ptr, thisWorker.index))
151     val leftsub = findWork(node.left)
152     if (leftsub ≠ ⊥) leftsub
153     else findWork(node.right)
154   else
155     val rightsub = findWork(node.right)
156     if (rightsub ≠ ⊥) rightsub
157     else findWork(node.left)
```

Figure 5.10: **RandomWalk** Strategy

Completely Random Strategies

Building on the randomization idea, we introduce an additional strategy called **RandomAll** where the traversal order in `findWork` is completely randomized. This strategy also randomizes all the other choices that the stealer and the victim make. Both the tree traversal order and the node chosen after the steal are thus changed in `findWork`. We show it in Figure 5.9 on the right.

In the **RandomWalk** strategy we only change the tree traversal order that the stealer does when searching for work and leave the rest of the choices fixed – victim picks the left node after expansion and the stealer picks the right node. The code is shown in Figure 5.10.

However, these completely random strategies result in a lower throughput and bigger tree sizes. Additionally randomizing the choice in lines 77 and 60 (**RandomAll**) is even less helpful, since the stealer and the victim clash immediately after steal more often.

```
def search(p: Ptr): Ptr =
  if (isLeaf(p.child)) p
  else
    val lp = search(p.child.left)
    val rp = search(p.child.right)
    val l = READ(lp.child)
    val r = READ(rp.child)
    if (remains(l) > remains(r)) l
    else r

def remains(n: Node) =
  n.until - READ(n.progress)
```

```
def findWork(ptr: Ptr): Node =
  val maxp = search(tree)
  val max = READ(maxp.child)
  if (remains(max) > 0)
    if (tryOwn(max)) max
    else if (¬isEligible(max)) ⊥
    else if (trySteal(maxp))
      val subnode = READ(maxp.right.child)
      if (tryOwn(subnode)) subnode
      else findWork(ptr)
    else findWork(ptr)
  else ⊥
```

Figure 5.11: **FindMax** Strategy

FindMax Strategy

The results of the five different strategies mentioned so far lead to the following observation. If a randomized strategy like **RandomWalk** or **AssignTop** works better than a suboptimal strategy like **Predefined** then some of its random choices help reduce the overall execution time and some increase it. Sometimes random decisions are good, but sometimes they are detrimental. So, there must exist an even better strategy which only makes the choices that lead to a better execution time.

Rather than providing a theoretical background for such a strategy, we propose a particular one which seems intuitive. Let workers traverse the entire tree and pick a node with most work, only then attempting to own or steal it. We call this strategy **FindMax**. Note that this cannot be easily implemented atomically. By the time that the work-stealing tree is completely traversed, the remaining work in each of the nodes will probably change since the last read. Still, an implementation that is only accurate given quiescence still serves as a decent heuristic for finding the node with most work.

The decisions about which node the victim and the stealer take after expansion remain the same as in the basic algorithm from Figure 5.5. We show the pseudocode for **FindMax** in Figure 5.11. This strategy yields an average tree size of 42 at $P = 8$, as well as a slightly better throughput.

The diagrams in Figure 5.12 reveal the postulated inverse correlation between the tree size and total execution time, both for the Intel i7-2600 and the Sun UltraSPARC T2 processor, which is particularly noticeable for **Assign** when the total number of workers is not a power of two. For some P **RandomAll** works slightly better than **FindMax** on UltraSPARC, but both are much more efficient than static batching, which deteriorates heavily once P exceeds the number of cores.

The **FindMax** strategy has an obvious downside that finding work requires traversing the entire work-stealing tree, which is a potential bottleneck. This problem can be addressed either by sampling the tree, or by maintaining a horizontally scalable eventually consistent

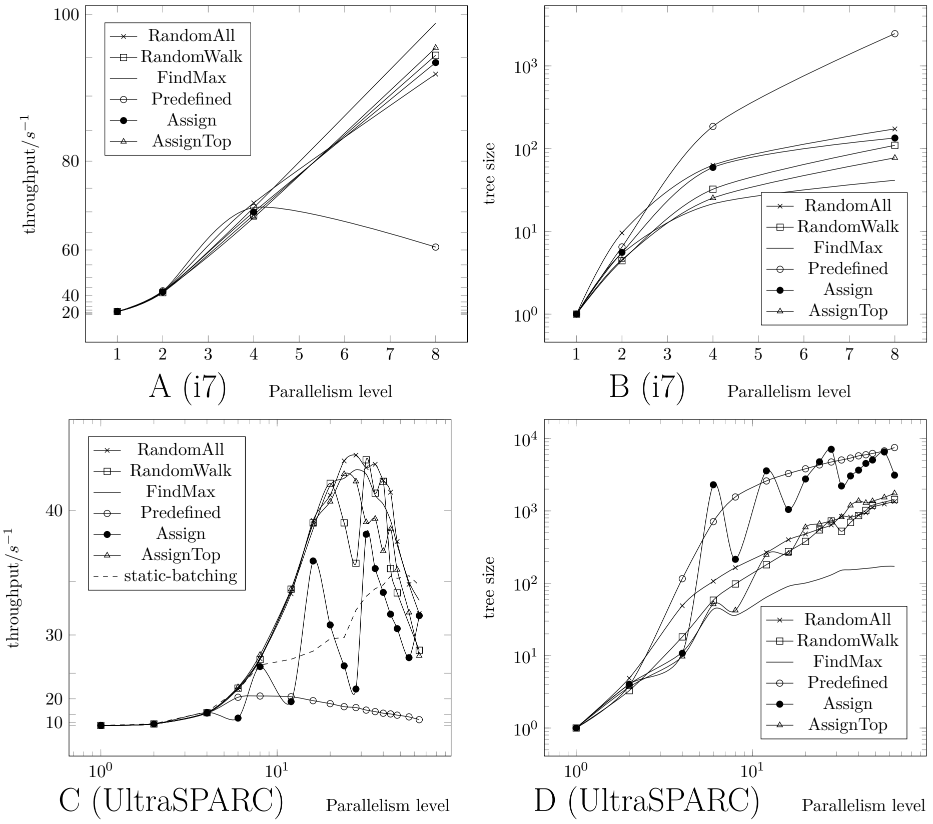


Figure 5.12: Comparison of `findWork` Implementations

priority queue that the workers update in order to have quick information about which nodes have a lot of work left. The need for this does not seem to exist on typical desktop machines yet.

5.2.5 Work-Stealing Node Batching Schedules

The results from the previous sections show that C1 can be fulfilled with a proper choice of node search. We focus on the C2 and C3 next by changing the workloads, namely the `kernel` method. Figures 5.13, 5.14 show a comparison of the work-stealing tree and some traditional data-parallel schedulers on a range of different workloads. Each workload pattern is illustrated prior to its respective diagrams, along with corresponding real-world examples. To avoid memory access effects and additional layers of abstraction each workload is minimal and synthetic, but corresponds to a practical use-case. To test C3, in Figure 5.13-5,6 we decrease the number of elements to 16 and increase the workload heavily. Fixed-size batching fails utterly for these workloads – the total number of elements is on the order of or well below the required `STEP` value. These workloads obviously require smaller `STEP` sizes to allow stealing, but that would break the baseline performance constraint, and the data-parallel scheduler is unable to distinguish the two types of workloads.

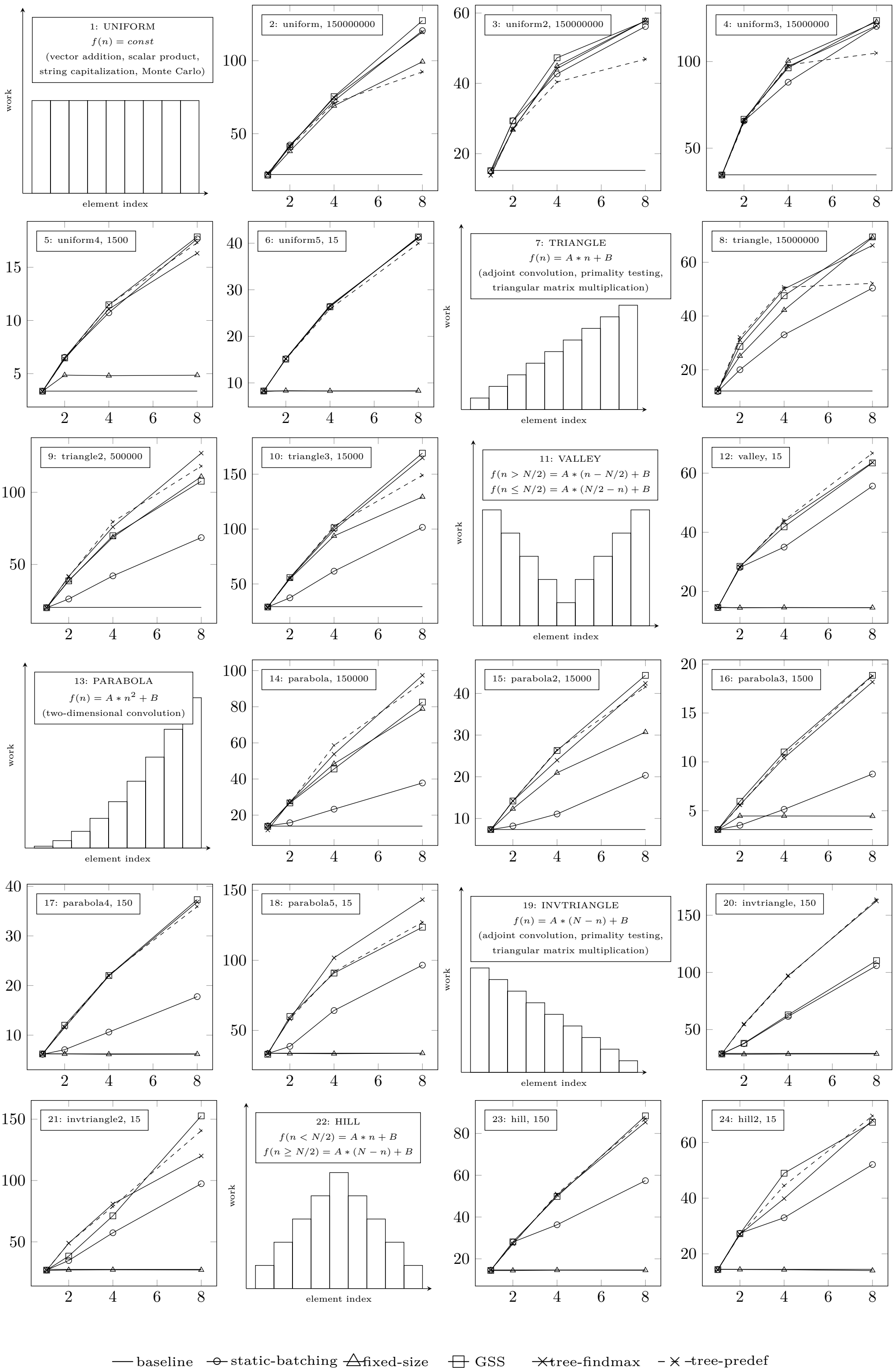


Figure 5.13: Comparison of `kernel` Functions I (Throughput/s^{-1} vs. #Workers)

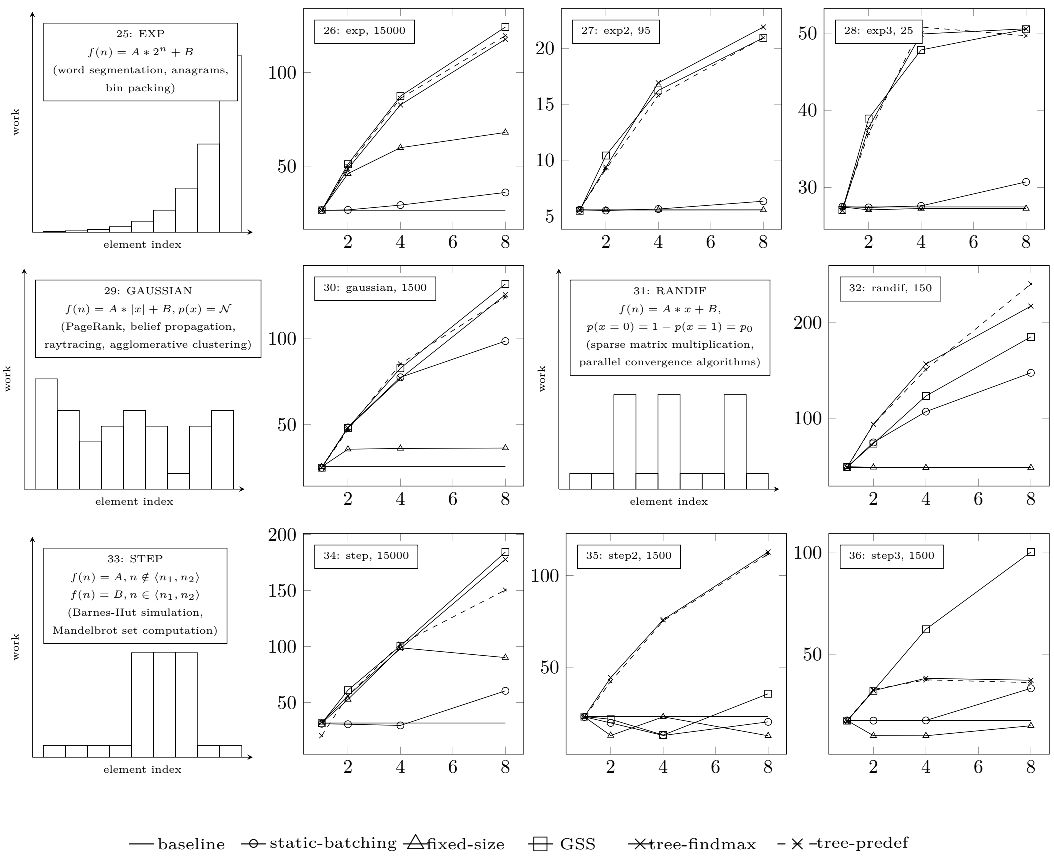


Figure 5.14: Comparison of `kernel` Functions II (Throughput/s^{-1} vs. #Workers)

The solution to this conundrum is that the `STEP` does not have to be a constant value. The worker owning a work-stealing node can change the size of the batch between the subsequent `tryAdvance` calls. Since there are many such calls between the time the work-stealing node becomes owned and the time it is completed or stolen, the sufficient requirement for the baseline workload costs to amortize the scheduling costs is that the *average* `STEP` size is above some threshold value. We will call the sequence of `STEP` values that are passed to the `tryAdvance` calls on a specific work-stealing node *the batching scheduler*. To solve the issues with coarse-grained workloads, the default work-stealing node batching schedule is *exponential* – the batch size is doubled from 1 up to some maximum value.

We thus modify the work-stealing tree in the following way. A mutable `step` field is added to `Node`, which is initially 1 and does not require atomic access. At the end of the `while` loop in the `workOn` method the `step` is doubled unless greater than some value `MAXSTEP`. As a result, workers start processing each node by cautiously checking if they can complete a bit of work without being stolen from and then increase the `step` exponentially. This naturally slows down the overall baseline execution, so we expect the `MAXSTEP` value to be greater than the previously established `STEP`. Indeed, on the i7-2600, we had to set `MAXSTEP` to 256 to maintain the baseline performance and at $P = 8$ even 1024. With these modifications work-stealing tree yields linear speedup for all uniform workloads.

Triangular workloads such as those shown in Figures 5.13-8,9,10 show that static batching can yield suboptimal speedup due to the uniform workload assumption. Figure 5.13-20 shows the inverse triangular workload and its negative effect on guided self-scheduling – the first-arriving processor takes the largest batch of work, which incidentally contains most work. We do not invert the other increasing workloads, but stress that it is neither helpful nor necessary to have batches above a certain size.

Figure 5.14-28 shows an exponentially increasing workload, where the work associated with the last element equals the rest of the work – the best possible speedup is 2. Figures 5.14-30,32 show two examples where a probability distribution dictates the workload, which occurs often in practice. Guided self-scheduling works well when the distribution is relatively uniform, but fails to achieve optimal speedup when only a few elements require more computation, for reasons mentioned earlier.

In the STEP distributions all elements except those in some range $\langle n_1, n_2 \rangle$ are associated with a very low amount of work. The range is set to 25% of the total number of elements. When its absolute size is above `MAXSTEP`, as in Figure 5.14-34, most schedulers do equally well. However, not all schedulers achieve optimal speedup as we decrease the total number of elements N and the range size goes below `MAXSTEP`. In Figure 5.14-35 we set $n_1 = 0$ and $n_2 = 0.25N$. Schedulers other than the work-stealing tree achieve almost no speedup, each for the same reasons as before. However, in Figure 5.14-36, we set $n_1 = 0.75N$ and $n_2 = N$ and discover that the work-stealing tree achieves a suboptimal speedup. The reason is the exponential batch increase – the first worker acquires a root node and quickly processes the cheap elements, having increased the batch size to `MAXSTEP` by the time it reaches the expensive ones. The real work is thus claimed by the first worker and the others are unable to acquire it. Assuming some batches are smaller and some larger as already explained, this problem cannot be worked around by a different batching order – there always exists a workload distribution such that the expensive elements are in the largest batch. In this adversarial setting the existence of a suboptimal work distribution for every batching order can only be overcome by randomization. We explore how to randomize batching in the Appendix D, showing how to improve the expected speedup for the problematic workloads.

5.2.6 Work-Stealing Combining Tree

As mentioned, the work-stealing tree is a particularly effective data-structure for a reduce operation. Parallel reduce is useful in the context of many other operations, such as finding the first element with a given property, finding the greatest element with respect

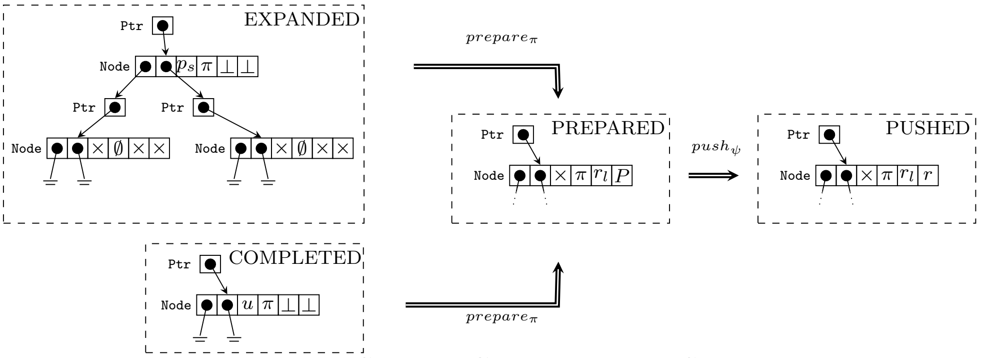


Figure 5.15: Work-Stealing Combining Tree State Diagram

to some ordering, filtering elements with a given property or computing an aggregate of all the elements (e.g. a sum).

There are two reasons why the work-stealing tree is amenable to implementing reductions. First, it preserves the order in which the work is split between processors, which allows using non-commutative operators for the reduce (e.g. computing the resulting transformation from a series of affine transformations can be parallelized by multiplying a sequence of matrices – the order is in this case important). Second, the reduce can largely be performed in parallel, due to the structure of the tree.

In this section, we describe the lock-free approach to combining intermediate results in the work-stealing tree. An advantage of this approach is increased throughput, as either of the child nodes can push the combined value to its parent. As we will see, one disadvantage is that the reduction operation may be repeated more than once in a lock-free reduction tree. Unless the reduction operation is non-side-effecting or idempotent, this can affect user programs. In such cases, we need to use locks – for example, the `combine` method on combiners in ScalaBlitz internally does locking to protect access to combiner state.

The work-stealing tree reduction is similar to the combining tree [Herlihy and Shavit(2008)], but it can proceed in a lock-free manner after all the node owners have completed their work, as we describe next. The general idea is to save the aggregated result in each node and then push the result further up the tree. Note that we did not save the return value of the `kernel` method in line 44 in Figure 5.5, making the scheduler applicable only to parallelizing `for` loops. Thus, we add a local variable `sum` and update it each time after calling `kernel`. Once the node ends up in a COMPLETED or EXPANDED state, we assign it the value of `sum`. Note that updating an invocation-specific shared variable instead would not only break the commutativity, but also lead to the same bottleneck as we saw before with fixed-size batching. We therefore add two new fields with atomic access to `Node`, namely `lresult` and `result`. We also add a new field `parent` to `Ptr`. We expand the set of abstract node states with two additional ones, namely PREPARED and PUSHED. The expanded state diagram is shown in Figure 5.15.

The parent field in Ptr is not shown in the diagram in Figure 5.15. The first two boxes in Node denote the left and the right child, respectively, as before. We represent the iteration state (progress) with a single box in Node. The iterator may either be stolen (p_s) or completed (u), but this is not important for the new states – we denote all such entries with ×. The fourth box represents the owner, the fifth and the sixth fields lresult and result. Once the work on the node is effectively completed, either due to a steal or a normal completion, the node owner π has to write the value of the sum variable to lresult. After doing so, the owner announces its completion by atomically writing a special value P to result, and by doing so pushes the node into the PREPARED state – we say that the owner prepares the node. At this point the node contains all the information necessary to participate in the reduction. The sufficient condition for the reduction to start is that the node is a leaf or that the node is an inner node and both its children are in the PUSHED state. The value lresult can then be combined with the result values of both its children and written to the result field of the node. Upon writing to the result field, the node goes into the PUSHED state. This push step can be done by any worker ψ and assuming all the owners have prepared their nodes, the reduction is lock-free. Importantly, the worker that succeeds in pushing the result must attempt to repeat the push step in the parent node. This way the reduction proceeds upwards in the tree until reaching the root. Once some worker pushes the result to the root of the tree, it notifies that the operation was completed, so that the thread that invoked the operation can proceed, in case that the parallel operation is synchronous. Otherwise, a future variable can be completed or a user callback invoked.

Before presenting the pseudocode, we formalize the notion of the states we described. In addition to the ones mentioned earliear, we identify the following new invariants.
INV6 Field n.lresult is set to $\perp$ when created. If a worker π overwrites the value $\perp$ of the field n.lresult then n.owner $= \pi$ and the node n is either in the EXPANDED state or the COMPLETED state. That is the last write to n.lresult.
INV7 Field n.result is set to $\perp$ when created. If a worker π overwrites the value $\perp$ of the field n.result with $\mathbb{P}$ then n.owner $= \pi$, the node n was either in the EXPANDED state or the COMPLETED state and the value of the field n.lresult is different than $\perp$. We say that the node goes into the PREPARED state.
INV8 If a worker ψ overwrites the value $\mathbb{P}$ of the field n.result then the node n was in the PREPARED state and was either a leaf or its children were in the PUSHED state. We say that the node goes into the PUSHED state.

We modify workOn so that instead of lines 47 through 51, it calls the method complete passing it the sum argument and the reference to the subtree. The pseudocodes for complete and an additional method pushUp are shown in Figure 5.16.

Upon completing the work, the owner checks whether the subtree was stolen. If so, it helps expand the subtree (line 186), reads the new node and writes the sum into lresult. After that, the owner pushes the node into the PREPARED state in line 192, retrying in

```
182 def complete(sum: T, tree: Ptr) =
183   node = READ(tree.child)
184   stolen = READ(node.progress) < 0
185   if (stolen)
186     trySteal(tree)
187     node = READ(tree.child)
188     node.lresult = sum
189   else
190     node.lresult = sum
191   while (READ(node.result) == ⊥)
192     CAS(node.result, ⊥, ℙ)
193   pushUp(tree)
194   ¬stolen
195
196 def pushUp(tree: Ptr): Unit =
197   node = READ(three.child)
198   res0 = READ(node.result)
199   if (res0 == ⊥)
200     // not yet in PREPARED state
201   else if (res0 ≠ ℙ)
202     // already in PUSHED state
203   else
204     res = ⊥
205     if (isLeaf(node)) res = lresult
206     else
207       left = READ(node.left.child)
208       right = READ(node.right.child)
209       rl = READ(left.result)
210       rr = READ(right.result)
211       if (rl ≠ ⊥ ∧ rr ≠ ⊥)
212         res = lresult + rl + rr
213     if (res ≠ ⊥)
214       if (CAS(node.result, res0, res))
215         if (tree.parent ≠ ⊥)
216           pushUp(tree.parent)
217         else tree.synchronized
218           { tree.notifyAll() }
219       else pushUp(tree)
```

Figure 5.16: Work-Stealing Combining Tree Pseudocode

the case of spurious failures, and calls `pushUp`.

The method `pushUp` may be invoked by the owner of the node attempting to write to the `result` field, or by another worker attempting to push the result up after having completed the work on one of the child nodes. The `lresult` field may not be yet assigned (line 200) if the owner has not completed the work – in this case the worker ceases to participate in the reduction and relies on the owner or another worker to continue pushing the result up. The same applies if the node is already in the PUSHED state (line 202). Otherwise, the `lresult` field can only be combined with the `result` values from the children if both children are in the PUSHED state. If the worker invoking `pushUp` notices that the children are not yet assigned the `result`, it will cease to participate in the reduction. Otherwise, it will compute the tentative result (line 212) and attempt to write it to `result` atomically with the CAS in line 214. A failed CAS triggers a retry, otherwise `pushUp` is called recursively on the parent node. If the current node is the root, the worker notifies any listeners that the final result is ready and the operations ends.

5.3 Speedup and Optimality Analysis

In Figure 5.14-36 we identified a workload distribution for which the work-stealing reduction tree had a particularly bad performance. This coarse workload consisted of a major prefix of elements which required a very small amount of computation followed by a minority of elements which required a large amount of computation. We call this workload coarse because the number of elements was on the order of magnitude of a certain value we called `MAXSTEP`.

To recap, the speedup was suboptimal due to the following. First, to achieve an optimal speedup for at least the baseline, not all batches can have fewer elements than a certain number. We have established this number for a particular architecture and environment, calling it `STEP`. Second, to achieve an optimal speedup for ranges the size of which is below `STEP·P`, some of the batches have to be smaller than the others. The technique we apply starts with a batch consisting of a single element and increases the batch size exponentially up to `MAXSTEP`. Third, there is no hardware interrupt mechanism available to interrupt a worker which is processing a large batch, and software emulations which consist of checking a volatile variable within a loop are too slow when executing the baseline. Fourth, the worker does not know the workload distribution and cannot measure time.

The above preconditions allow a single worker obtain the largest batch before the other workers had a chance to steal some work for a particular workload distribution. This is the case for any batching scheduler. Justifying this claim requires a set of more formal definitions. We start by defining the context in which the scheduler executes.

Oblivious conditions. If a data-parallel scheduler is unable to obtain information about the workload distribution, nor information about the amount of work it had previously executed, we say that the data-parallel scheduler works in *oblivious conditions*.

Assume that a worker decides on some batching schedule $c_1, c_2, \ldots, c_k$ where c_j is the size of the j-th batch and $\sum_{j=1}^{k} c_j = N$, where N is the size of the range. No batch is empty, i.e. $c_j \neq 0$ for all j. In oblivious conditions the worker does not know if the workload resembles the baseline mentioned earlier, so it must assume that it does and minimize the scheduling overhead. The baseline is not only important from a theoretical perspective being one of the potentially worst-case workload distribution, but also from a practical one – in many problems parallel loops have a uniform workload. We now define what this baseline means more formally.

The baseline constraint. Let the workload distribution be a function $w(i)$ which gives the amount of computation needed for range element i. We say that a data-parallel scheduler respects *the baseline constraint* if and only if the speedup s_p with respect to a sequential loop is arbitrarily close to linear when executing the workload distribution $w(i) = w_0$, where w_0 is the minimum amount of work needed to execute a loop iteration.

Arbitrarily close here means that ϵ in $s_p = \frac{P}{1+\epsilon}$ can be made arbitrarily small.

The baseline constraint tells us that it may be necessary to divide the elements of the loop into batches, depending on the scheduling (that is, communication) costs. As we have seen in the experiments, while we should be able to make the ϵ value arbitrarily small, in practice it is small enough when the scheduling overhead is no longer observable in the measurement. Also, we have shown experimentally that the average batch size should be bigger than some value in oblivious conditions, but we have used particular scheduler instances. Does this hold in general, for every data-parallel scheduler? The answer is yes, as we show in the following lemma.

Lemma 5.1 *If a data-parallel scheduler that works in oblivious conditions respects the baseline constraint then the batching schedule* $c_1, c_2, \dots, c_k$ *is such that:*

$$\frac{\sum_{j=1}^{k} c_j}{k} \geq S(\epsilon) \tag{5.2}$$

Proof. The lemma claims that in oblivious conditions the average batch size must be above some value which depends on the previously defined ϵ, otherwise the scheduler will not respect the baseline constraint.

The baseline constraint states that $s_p = \frac{P}{1+\epsilon}$, where the speedup s_p is defined as T_0/T_p, where T_0 is the running time of a sequential loop and T_p is the running time of the scheduler using P processors. Furthermore, $T_0 = T \cdot P$ where T is the optimal parallel running time for P processors, so it follows that $\epsilon \cdot T = T_p - T$. We can also write this as $\epsilon \cdot W = W_p - W$. This is due to the running time being proportionate to the total amount of executed work, whether scheduling or useful work. The difference $W_p - W$ is exactly the scheduling work W_s, so the baseline constraint translates into the following inequality:

$$W_s \leq \epsilon \cdot W \tag{5.3}$$

In other words, the scheduling work has to be some fraction of the useful work. Assuming that there is a constant amount of scheduling work W_c per every batch, we have $W_s = k \cdot W_c$. Lets denote the average work per element with $\overline{w}$. We then have $W = N \cdot \overline{w}$. Combining these relations we get $N \geq k \cdot \frac{W_c}{\epsilon \cdot \overline{w}}$, or shorter $N \geq k \cdot S(\epsilon)$. Since N is equal to the sum of all batch sizes, we derive the following constraint:

$$\frac{\sum_{j=1}^{k} c_j}{k} \geq \frac{W_c}{\epsilon \cdot \overline{w}} \tag{5.4}$$

In other words, the average batch size must be greater than some value $S(\epsilon)$ which depends on how close we want to get to the optimal speedup. Note that this value is inversely proportionate to the average amount of work per element $\overline{w}$ – the scheduler could decide more about the batch sizes if it knew something about the average workload, and grows with the scheduling cost per batch W_c – this is why it is especially important to make the `workOn` method efficient. We already saw the inverse proportionality with ϵ in Figure 5.7. In part, this is why we had to make `MAXSTEP` larger than the chosen `STEP` (we also had to increase it due to increasing the scheduling work in `workOn`, namely, W_c). This is an additional constraint when choosing the batching schedule.

With this additional constraint there always exists a workload distribution for a given batching schedule such that the speedup is suboptimal, as we show next.

Lemma 5.2 *Assume that $S(\epsilon) > 1$, for the desired ϵ. For any fixed batching schedule $c_1, c_2, \ldots, c_k$ there exists a workload distribution such that the scheduler executing it in oblivious conditions yields a suboptimal schedule.*

Proof. First, assume that the scheduler does not respect the baseline constraint. The baseline workload then yields a suboptimal speedup and the statement is trivially true because $S(\epsilon) > 1$.

Otherwise, assume without the loss of generality that at some point in time a particular worker ω is processing some batch c_m the size of which is greater or equal to the size of the other batches. This means the size of c_m is greater than 1, from the assumption. Then we can choose a workload distribution such that the work $W_m = \sum_{i=N_m}^{N_m+c_m} w(i)$ needed to complete batch c_m is arbitrarily large, where $N_m = \sum_{j=1}^{m-1} c_j$ is the number of elements in the batching schedule coming before the batch c_m. For all the other elements we set $w(i)$ to be some minimum value w_0. We claim that the obtained speedup is suboptimal. There is at least one different batching schedule with a better speedup, and that is the schedule in which instead of batch c_m there are two batches c_{m_1} and c_{m_2} such that c_{m_1} consists of all the elements of c_m except the last one and c_{m_2} contains the last element. In this batching schedule some other worker can work on c_{m_2} while ω works on c_{m_1}. Hence, there exists a different batching schedule which leads to a better speedup, so the initial batching schedule is not optimal.

We can ask ourselves what is the necessary condition for the speedup to be suboptimal. We mentioned that the range size has to be on the same order of magnitude as S above, but can we make this more precise? We could simplify this question by asking what is the necessary condition for the worst-case speedup of 1 or less. Alas, we cannot find necessary conditions for all schedulers because they do not exist – there are schedulers which do not need any preconditions in order to consistently produce such a speedup (think of a sequential loop or, worse, a "scheduler" that executes an infinite loop). Also, we already saw that a suboptimal speedup may be due to a particularly bad workload distribution, so maybe we should consider only particular distributions, or have some conditions on them. What we will be able to express are the necessary conditions on the range size for the the existence of a scheduler which achieves a speedup greater than 1 on any workload. Since the range size is the only information known to the scheduler in advance, it can be used to affect its decisions in a particular implementation.

The worst-case speedups we saw occurred in scenarios where one worker (usually the invoker) started to work before all the other workers. To be able to express the desired conditions, we model this delay with a value T_d.

Lemma 5.3 *Assume a data-parallel scheduler that respects the baseline constraint in oblivious conditions. There exists some minimum range size N_1 for which the scheduler can yield a speedup greater than* 1 *for any workload distribution.*

Proof. We first note that there is always a scheduler that can achieve the speedup 1, which is merely a sequential loop. We then consider the case when the scheduler is parallelizing the baseline workload. Assume now that there is no minimum range size N_1 for which the claim is true. Then for any range size N we must be able to find a range size $N + K$ such that the scheduler still cannot yield speedup 1 or less, for a chosen K. We choose $N = \frac{f \cdot T_d}{w_0}$, where w_0 is the amount of work associated with each element in the baseline distribution and f is an architecture-specific constant describing the computation speed. The chosen N is the number of elements that can be processed during the worker wakeup delay T_d. The workers that wake up after the first worker ω processes N elements have no more work to do, so the speedup is 1. However, for range size $N + K$ there are K elements left that have not been processed. These K elements could have been in the last batch of ω. The last batch in the batching schedule chosen by the scheduler may include the Nth element. Note that the only constraint on the batch size is the lower bound value $S(\epsilon)$ from Lemma 5.1. So, if we choose $K = 2S(\epsilon)$ then either the last batch is smaller than K or is greater than K. If it is smaller, then a worker different than ω will obtain and process the last batch, hence the speedup will be greater than 1. If it is greater, then the worker ω will process the last batch – the other workers that wake up will not be able to obtain the elements from that batch. In that case there exists a better batching order which still respects the baseline constraint and that is to divide the last batch into two equal parts, allowing the other workers to obtain some work and yielding a speedup greater than 1. This contradicts the assumption that there is no minimum range size N_1 – we know that N_1 is such that:

$$\frac{f \cdot T_d}{w_0} \leq N_1 \leq \frac{f \cdot T_d}{w_0} + 2 \cdot S(\epsilon) \tag{5.5}$$

Now, assume that the workload $w(i)$ is not the baseline workload w_0. For any workload we know that $w(i) \geq w_0$ for every i. The batching order for a single worker has to be exactly the same as before due to oblivious conditions. As a result the running time for the first worker ω until it reaches the Nth element can only be larger than that of the

```
def work(it: StealIterator[T]) =
  var step = 0
  var res = zero
  while (it.state() == 𝔸)
    step = update(step)
    val batch = it.nextBatch(step)
    if (batch >= 0)
      res = combine(res, apply(it, batch))
  it.result = res
```

Figure 5.17: The Generalized `workOn` Method

baseline. This means that the other workers will wake up by the time ω reaches the Nth element, and obtain work. Thus, the speedup can be greater than 1, as before.

We have so far shown that we can decide on the average batch size if we know something about the workload, namely, the average computational cost of an element. We have also shown when we can expect the worst case speedup, potentially allowing us to take prevention measures. Finally, we have shown that any data-parallel scheduler deciding on a fixed schedule in oblivious conditions can yield a suboptimal speedup. Note the wording “fixed” here. It means that the scheduler must make a definite decision about the batching order without any knowledge about the workload, and must make the same decision every time – it must be deterministic. As hinted before, the way to overcome an adversary that is repetitively picking the worst case workload is to use randomization when producing the batching schedule.

5.4 Steal-Iterators – Work-Stealing Iterators

The goal of this section is to augment the *iterator* abstraction with the facilities that support work-stealing, in the similar way the iterators were enriched with the `split` method in Chapter 2. This will allow using the work-stealing tree data-parallel scheduler with any kind of data structure and not just parallel loops. The previously shown `progress` value served as a placeholder for all the work-stealing information in the case of parallel loops.

There are several parts of the work-stealing scheduler that we can generalize. We read the value of `progress` in line 40 to see if it is negative (indicating a steal) or greater than or equal to `until` (indicating that the loop is completed) in line 41.

Here the value of `progress` indicates the `state` the iterator is in – either available ($\mathbb{A}$), stolen ($\mathbb{S}$) or completed ($\mathbb{C}$). In line 14 we atomically update `progress`, consequently deciding on the number of elements that can be processed. This can be abstracted away with a method `nextBatch` that takes a desired number of elements to traverse and returns an estimated number of elements to be traversed, or -1 if there are none left. Figure 5.17 shows an updated version of the loop scheduling algorithm that relies on these methods. Iterators should also abstract the method `markStolen` shown earlier.

We show the complete work-stealing iterator interface in Figure 5.18. The additional method `owner` returns the index of the worker owning the iterator. The method `next` can be called as long as the method `hasNext` returns `true`, just as with the ordinary iterators. Method `hasNext` returns `true` if `next` can be called before the next `nextBatch` call. Finally, the method `split` can only be called on $\mathbb{S}$ iterators and it returns a pair of iterators such that the disjoint union of their elements are the remaining elements of the original iterator. This implies that `markStolen` must internally encode the iterator state immediately when it gets stolen.

The contracts of these methods are formally expressed below. We implicitly assume termination and a specific iterator instance. Unless specified otherwise, we assume linearizability. When we say that a method M is owner-specific (π-specific), it means that every invocation by a worker π is preceeded by a call to `owner` returning π. For non-owner-specific M `owner` returns $\psi \neq \pi$.

Contract `owner`. If an invocation returns π at time t_0, then $\forall t_1 \geq t_0$ invocations return π.

Contract `state`. If an invocation returns $s \in \{\mathbb{S}, \mathbb{C}\}$ at time t_0, then all invocations at $t \geq t_0$ return s, where C and S denote completed and stolen states, respectively.

Contract `nextBatch`. If an invocation exists at some time t_0 then it is π-specific and the parameter `step` ≥ 0. If the return value c is -1 then a call to `state` at $\forall t_1 > t_0$ returns $s \in \{\mathbb{S}, \mathbb{C}\}$. Otherwise, a call to `state` at $\forall t_{-1} < t_0$ returns $s = A$, where A is the available state.

Contract `markStolen`. Any invocations at t_0 is non-owner-specific and every call to `state` at $t_1 > t_0$ returning $s \in \{\mathbb{S}, \mathbb{C}\}$.

Contract `next`. A non-linearizable π-specific invocation is linearized at t_1 if there is a `hasNext` invocation returning `true` at $t_0 < t_1$ and there are no `nextBatch` and `next` invocations in the interval $\langle t_0, t_1 \rangle$.

Contract `hasNext`. If a non-linearizable π-specific invocation returns `false` at t_0 then all `hasNext` invocations in $\langle t_0, t_1 \rangle$ return `false`, where there are no `nextBatch` calls in $\langle t_0, t_1 \rangle$.

Contract `split`. If an invocation returns a pair (n_1, n_2) at time t_0 then the call to `state` returned $\mathbb{S}$ at some time $t_{-1} < t_0$.

Traversal contract. Define $\overline{X} = x_1 x_2 \ldots x_m$ as the sequence of return values of `next` invocations at times $t'_1 < t'_2 < \ldots < t'_m$. If a call to `state` at $t > t'_m$ returns $\mathbb{C}$ then $e(i) = \overline{X}$. Otherwise, let an invocation of `split` on an iterator i return (i_1, i_2). Then $e(i) = \overline{X} \cdot e(i_1) \cdot e(i_2)$, where $\cdot$ is concatenation. There exists a fixed E such that $E = e(i)$

```
StealIterator[T] {
  def owner(): Int
  def state(): 𝔸 ∨ 𝕊 ∨ ℂ
  def nextBatch(step: Int): Int
  def markStolen(): Unit
  def hasNext: Boolean
  def next(): T
  def split(): (StealIterator[T], StealIterator[T])
}
```

Figure 5.18: The `StealIterator` Interface

for all valid sequences of `nextBatch` and `next` invocations.

The last contract states that every iterator always traverses the same elements in the same order. Having formalized the work-stealing iterators, we show several concrete implementations.

5.4.1 Indexed Steal-Iterators

This is a simple iterator implementation following from refactorings in Figure 5.17. It is applicable to parallel ranges, arrays, vectors and data-structures where indexing is fast. The implementation for ranges in Figure 5.19 uses the `private` keyword for the fields `nextProgress` and `nextUntil` used in `next` and `hasNext`. Since their contracts ensure that only the owner calls them, their writes need not be globally visible and are typically faster. The field `progress` is marked with the keyword `atomic`, and is modified by the CAS in the line 254, ensuring that its modifications are globally visible through a memory barrier.

All method contracts are straightforward to verify and follow from the linearizability of CAS. For example, if `state` returns $\mathbb{S}$ or $\mathbb{C}$ at time t_0, then the `progress` was either negative or equal to `until` at t_0. All the writes to `progress` are CAS instructions that check that `progress` is neither negative nor equal to `until`. Therefore, `progress` has the same value $\forall t > t_0$ and `state` returns the same value $\forall t > t_0$ (contract *state*).

5.4.2 Hash Table Steal-Iterators

The implementation of work-stealing iterators for flat hash-tables we show in this section is similar to the iterators for data-structures with fast indexing. Thus, the iteration state can still be represented with a single integer field `progress`, and invalidated with `markStolen` in the same way as with `IndexIterator`. The `nextBatch` has to compute the expected number of elements between to array entries using the load factor `lf` as

```
RangeIterator implements StealIterator[Int] {
  private var nextProgress = -1
  private var nextUntil = -1
  atomic var progress: Int
  val owner: Int
  val until: Int
  def state() =
    val p = READ(progress)
    if (p ≥ until) return ℂ
    else if (p < 0) return 𝕊
    else return 𝔸
  def nextBatch(s: Int): Int =
    if (state() ≠ 𝔸) return -1
    else
      val p = READ(progress)
      val np = math.min(p + s, until)
      if (¬CAS(progress, p, np)) return nextBatch(s)
      else
        nextProgress = p
        nextUntil = np
        return np - p
  def markStolen() =
    val p = READ(progress)
    if (p < until ∧ p ≥ 0)
      if (¬CAS(progress, p, -p - 1)) markStolen()
  def hasNext = return nextProgress < nextUntil
  def next() =
    nextProgress += 1
    return nextProgress - 1
}
```

Figure 5.19: The `IndexIterator` Implementation

follows:

```
def nextBatch(step: Int): Int = {
  val p = READ(progress)
  val np = math.min(p + (step / lf).toInt, until)
  if (¬CAS(progress, p, np)) return nextBatch(step)
  else
    nextProgress = p; nextUntil = np; return np - p
}
```

We change the `next` and `hasNext` implementations so that they traverse the range between `nextProgress` and `nextUntil` as a regular single-threaded hash-table iterator implementation. This implementation relies on the hashing function to achieve good load-balancing, which is common with hash-table operations.

5.4.3 Tree Steal-Iterators

A considerable number of applications use the tree representation for their data. Text editing applications often represent text via ropes, and HTML document object model is based on an n-ary tree. Ordered sets are usually implemented as balanced search trees, and most priority queues are balanced trees under the hood. Persistent hash tables present in many functional languages are based on hash tries. Parallelizing operations on trees is thus a desirable goal.

```
TreeIterator[T] extends StealIterator[T] {
  private val localstack: Array[Tree]
  private var depth: Int
  atomic var stack: Bitset

  def state() =
    val s = READ(stack)
    return s & 0x3

  def markStolen() =
    val s = READ(stack)
    if (s & 0x3 = 𝔸)
      val ns = (ns & ~0x3) | 𝕊
      if (¬CAS(stack, s, ns)) markStolen()

  def topBitset(s: Bitset) =
    val d = 2 * depth
    return (s & (0x3 << d)) >> d

  def top() = localstack[depth - 1]

  def pop(s: Bitset) =
    depth = depth - 1
    localstack(depth) = null
    return s & ~(0x3 << (2 + 2 * depth))

  def push(s: Bitset, v: Bitset, t: Tree)
    localstack(depth) = t
    depth = depth + 1
    return s | (v << (2 * depth))

  def switch(s: Bitset, v: Bitset) =
    val d = 2 * depth
    return (s & ~(0x3 << d)) | (v << d)
```

```
  def nextBatch(step: Int): Int = {
    val s = READ(stack)
    var ns = s
    var batchSize = -1
    if (s & 0x3 ≠ 𝔸) return -1 else
      val tm = topBitset(s)
      if (tm = 𝔹
        ∨ (tm = 𝕋 ∧ top().right.isLeaf))
        ns = pop(ns)
        while (topBitset(ns) = ℝ ∧ depth > 0)
          ns = pop(ns)
        if (depth = 0) ns = (ns & ~0x3) | ℂ
        else
          ns = switch(ns, 𝕋)
          batchSize = 1
          setNextValue(top().value)
      else if (tm = 𝕋)
        ns = switch(ns, ℝ)
        val n = top().right
        while (¬n.left.isLeaf
          ∧ bound(depth) ≥ step)
          ns = push(ns, 𝕃, n)
          n = n.left
        if (bound(depth) < step)
          ns = push(ns, 𝔹, n)
          batchSize = bound
          setNextSubtree(n)
        else
          ns = push(ns, 𝕋, n)
          batchSize = 1
          setNextValue(n.value)
    while (¬CAS(stack, s, ns))
      val ss = READ(stack)
      if (ss ∈ { 𝕊, ℂ }) return -1
    return batchSize
}
```

Figure 5.20: The `TreeIterator` Data-Type and Helper Methods

Stealers for flat data structures were relatively simple, but efficient lock-free tree steal-iterators are somewhat more involved. In this section we do not consider unbalanced binary trees since they cannot be efficiently parallelized – a completely unbalanced tree degrades to a linked list. We do not consider n-ary trees either, but note that n-ary tree steal-iterators are a straightforward extension to the steal-iterators presented in this section. We note that steal-iterators for trees in which every node contains the size of its subtree can be implemented similarly to `IndexIterator`s presented earlier, since their iteration state can be encoded as a single integer. Finally, iteration state for trees with pointers to parent nodes (two-way trees) can be encoded as a memory address of the current node, so their steal-iterators are trivial.

Therefore, in this section we show a steal-iterator for balanced binary search trees in which nodes do not have parent pointers and do not maintain size of their corresponding subtrees (one-way trees). In this representation each node is either an inner node containing a single data `value` and pointers to the `left` and the `right` child, or it is a leaf node (`isLeaf`), in which case it does not contain data elements. AVL trees, red-black trees and binary hash tries fit precisely this description.

The trees we examine have two important properties. First, given a node T at the depth d and the total number of keys N in the entire tree, we can always compute a bound on the depth of the subtree rooted at T from d and N. Similarly, we can compute a bound on the number of keys in the subtree of T. These properties follow from the fact that the depth of a balanced tree is bounded by its number of elements. Red-black trees, for example, have the property that their depth d is less than or equal to $2 \log N$ where N is the total number of keys.

Given a tree root, the iteration state can be encoded as a `stack` of decisions $\mathbb{L}$ and $\mathbb{R}$ that denote turning left or right at a given node. The top of the stack contains a terminal symbol $\mathbb{T}$ that denotes that the corresponding node is currently being traversed, or a symbol $\mathbb{B}$ indicating an entire subtree as a batch of elements that the steal-iterator last committed to process. Examples of tree iteration states are shown in Figure 5.21, where a worker first traverses a node C, proceeds by traversing a single node B and then decides to traverse the entire subtree D.

We represent `stack` with a bitset, using 2 bits to store single stack entry. The first 2 bits of this bitset have a special role – they encode one of the three steal-iterator states $\mathbb{A}$, $\mathbb{S}$ and $\mathbb{C}$. The stealers and workers update the steal-iterator state atomically by replacing the `stack` bitset with CAS instructions as shown in Figure 5.20. Stealers invoke `markStolen` that atomically changes the steal-iterator state bits in line 288, making sure that any subsequent `nextBatch` calls fail by invalidating their next CAS in line 340, which in turn requires the state bits to be equal to $\mathbb{A}$. Workers invoke `nextBatch` that atomically changes the currently traversed node or a subtree. A worker owning a particular steal-iterator additionally maintains the actual stack of tree nodes on the current traversal path in its local array `localstack`, whose size is bounded by the depth d of the corresponding tree. For convenience, it also maintains the current depth `depth`. Calling `nextBatch` starts by checking to see if the node is in the available state $\mathbb{A}$, and returning -1 if it is not. Part of the code between lines 314 and 339 is identical to that of a regular sequential tree iterator – it identifies the currently traversed node and replaces it, updating `localstack` and `depth` in the process. At each point it updates the tentative new state of the stack `ns` by adding and removing the symbols $\mathbb{L}$, $\mathbb{R}$, $\mathbb{B}$ and $\mathbb{T}$ using the helper `push`, `pop` and `switch` methods. Note that in line 328 the worker relies on the `bound` value at a given `depth` to estimate the number of elements in particular subtrees, and potentially decide on batching the elements. Calls to `setNextValue` and `setNextSubtree` set the next value or subtree to be traversed – they update the steal-iterator state so that the subsequent `next` and `hasNext` calls work correctly.

Once all the updates of the worker-local state are done, the worker attempts to atomically change the state of the `stack` with the new value `ns` in line 340, failing only if a concurrent

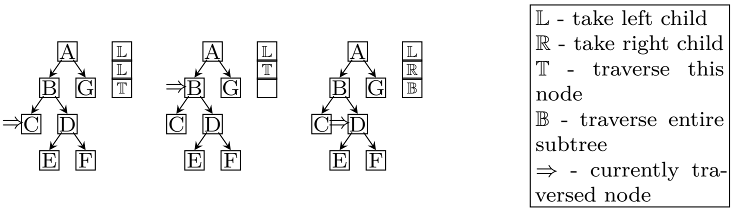


Figure 5.21: `TreeIterator` State Diagram

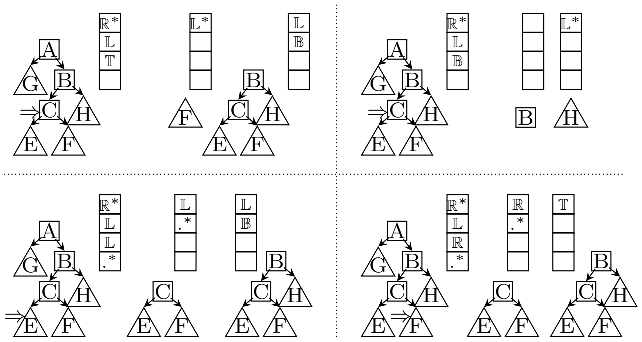


Figure 5.22: `TreeIterator` Splitting Rules

steal has occurred, in which case it returns -1.

We note that if the CAS allows 64-bit values then this approach only allows encoding steal-iterators for balanced trees that have less than 2^{16} elements. An assumption here is that the ratio of the longest and the shortest path from the root to the node is 2 (red-black and AVL trees being typical examples). Otherwise, encoding the state of the steal-iterator requires more bits, but must still be accessed atomically. A lightweight spin locks can protect the state of the bitset – although such steal-iterators are not lock-free anymore, the critical sections of their locks are short and workers searching for work do not need to block upon running on such a steal-iterator, as they can check if the steal-iterator is locked.

We do not show the pseudocode for splitting the steal-iterator after it has been marked stolen, but show the important classes of different states the steal-iterator can be in, in Figure 5.22. We encode the state of the stack with a regular expression of stack symbols – for example, the expression $\mathbb{R}^*\mathbb{L}\mathbb{T}$ means that the stack contains a sequence of right turns followed by a single left turn and then the decision $\mathbb{T}$ to traverse a single node. A steal-iterator in such a state should be split into two steal-iterators with states $\mathbb{L}^*$ and $\mathbb{R}^*\mathbb{L}\mathbb{B}$, as shown in Figure 5.22. The remaining states $\mathbb{R}^*\mathbb{B}$ and $\mathbb{R}^*\mathbb{T}$ are omitted for brevity.

5.5 Related Work

Per processor (henceforth, worker) work assignment done statically during compile time or linking, to which we will refer to as *static batching*, was studied extensively [Koelbel and Mehrotra(1991)] [Sarkar(2000)]. Static batching cannot correctly predict workload distributions for any problem, as shown by the second program in Figure 5.1. Without knowing the numbers in the set exactly, batches cannot be statically assigned to workers in an optimal way – some workers may end up with more work than the others. Still, although cost analysis is not the focus here, we advocate combining static analysis with runtime techniques.

To address the need for load balancing at runtime, work can be divided into a lot of small batches. Only once each worker processes its batch, it requests a new batch from a centralized queue. We will refer to this as *fixed-size batching* [Kruskal and Weiss(1985)]. In fixed-size batching the workload itself dictates the way how work is assigned to workers. This is a major difference with respect to static batching. In general, in the absence of information about the workload distribution, scheduling should be *workload-driven*. A natural question arises – what is the ideal size for a batch? Ideally, a batch should consist of a single element, but the cost of requesting work from a centralized queue is prohibitively large for that. For example, replacing the increment `i += 1` with an atomic CAS can increase the running time of a `while` loop by nearly a magnitude on modern architectures. The batch size has to be the least number of elements for which the cost of accessing the queue is amortized by the actual work. There are two issues with this technique. First, it is not scalable – as the number of workers increases, so does contention on the work queue (Figure 5.7). This requires increasing batch sizes further. Second, as the granularity approaches the batch size, the work division is not fine-grained and the speedup is suboptimal (Figure 5.13, where size is less than 1024).

Guided self-scheduling [Polychronopoulos and Kuck(1987)] solves some granularity issues by dynamically choosing the batch size based on the number of remaining elements. At any point, the batch size is R_i/P, where R_i is the number of remaining elements and P is the number of workers – the granularity becomes finer as there is less and less work. Note that the first-arriving worker is assigned the largest batch of work. If this batch contains more work than the rest of the loop due to irregularity, the speedup will not be linear. This is shown in Figures 5.13-20, 5.14-35. *Factoring* [Hummel et al.(1992)Hummel, Schonberg, and Flynn] and *trapezoidal self-scheduling* [Tzen and Ni(1993)] improve on guided-self scheduling, but have the same issue with those workload distributions.

One way to overcome the contention issues inherent to the techniques above is to use several work queues rather than a centralized queue. In this approach each processor starts with some initial work on its queue and commonly steals from other queues when it runs out of work – this is known as *work-stealing*, a technique applicable to both task- and data-parallelism. One of the first uses of work-stealing dates to the Cilk language [Blumofe et al.(1995)Blumofe, Joerg, Kuszmaul, Leiserson, Randall, and Zhou], in which processors relied on the fast and slow version of the code to steal stack frames from each other. Recent developments in the X10 language are based on similar techniques [Tardieu et al.(2012)Tardieu, Wang, and Lin]. Work-stealing typically relies on the use of work-stealing queues [Arora et al.(1998)Arora, Blumofe, and Plaxton] [Lea(2000)] and deques [Chase and Lev(2005)], implementations ranging from blocking to lock-free. In the past data-parallel collections frameworks relied on using task-parallel schedulers under the hood [Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky] [Reinders(2007)]. The work in this thesis shows that customizing work-stealing for data-parallelism allows more fine-grained stealing and superior load-balancing.

5.6 Conclusion

Although based on similar ideas, data-parallel work-stealing has a different set of assumptions compared to task-based work-stealing. Where task-based work-stealing schedulers rely on the fact that tasks can unpredictably spawn additional work, data-parallel work-stealing schedulers work with a predefined set of tasks, called elements. Here, the key to efficient load-balancing is combining elements into batches, with the goal of reducing the scheduling costs to a minimum. These batches need to be chosen carefully to reduce the risk of a computationally expensive batch being monopolized by a single processor. Simultaneously, different processors should decide on batching in isolation – the costs of synchronization should only be paid on rare occasions, when some processor needs to steal work. Work-stealing tree is a data structure designed to meet these goals.

Having shown that the work-stealing tree scheduler is superior to the classic batching techniques, and is able to parallelize irregular workloads particularly well, we ask ourselves – how useful is the work-stealing tree in practice? In a plethora of applications, workloads are relatively uniform and smooth, exhibiting triangular or parabolic patterns. As shown in Section 6.5, a task-based scheduler from Chapter 2 achieves acceptable performance for all but the most skewed inputs. Often, the gain is only $10\% - 20\%$ when using the work-stealing tree scheduler. Nevertheless, we argue that, in most managed runtimes, the work-stealing tree scheduling is preferable to task-based work-stealing, for multiple reasons.

The main reason for using the work-stealing tree scheduling is its simplicity. The basic work-stealing tree algorithm for parallel loops fits into just 80 lines of code, and is easy to understand. This is a small price to pay for a flexible scheduler that handles irregular workloads well.

The real complexity of the work-stealing tree scheduler is factored out into work-stealing iterators. While work-stealing iterator implementations for indexed data structures like arrays, hash-tables and size-combining trees, such as Conc-trees, are both straightforward and efficient, the lock-free steal-iterators for one-way trees, described in Section 5.4, are painfully hard to implement, and are more expensive. The good news is that we do not have to use them. We can either revert to lock-based tree steal-iterators, or just use regular splitters – nodes in the work-stealing tree can be expanded preemptively when using data structures for which stealing is problematic.

Although most workloads are not heavily irregular, the use of nested parallelism can make the workload arbitrarily skewed. In this case, a nested data-parallel operation creates a separate work-stealing tree, announces its existence to other workers, and starts working on it directly. Note that working on the tree, rather than waiting for the result of the data-parallel operation, prevents starvation. The caller thread can only be blocked in the work-stealing tree if some other thread is also working in the same work-stealing tree, indicating that the operation will eventually complete. To reduce the amount of blocking, work-stealing tree nodes can contain pointers to the work-stealing trees corresponding to the nested data-parallel operations.

Finally, the work-stealing tree serves as a reduction tree for the results. Combining is much simpler than efficiently combining the results from different tasks, on a platform that does not natively support work-stealing, and can be done in a lock-free manner.

6 Performance Evaluation

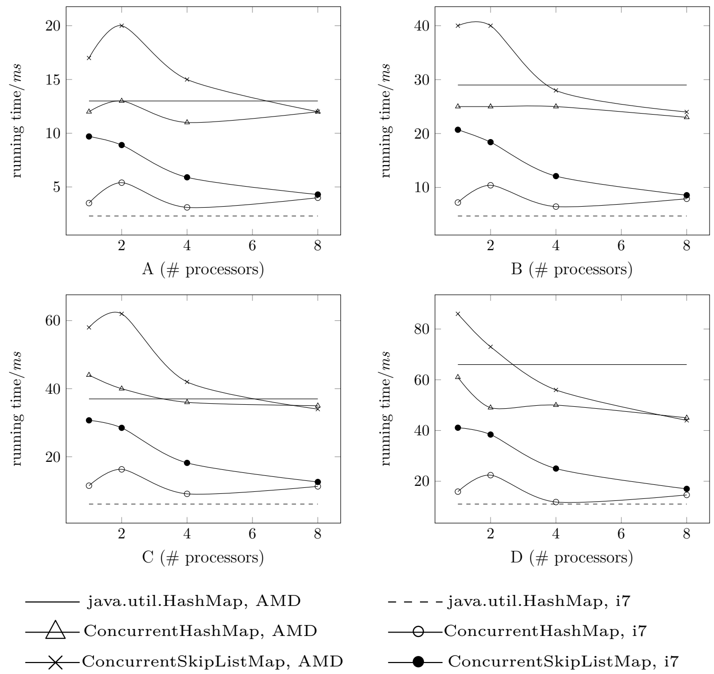
In this section we show experimental performance evaluations for algorithms and data structures introduced in the previous chapters. Sections 6.1 and 6.2 evaluate the performance of the basic data-parallel collections shown in Chapter 2. Section 6.3 compares the efficiency of various Conc-tree operations against similar data structures. and Section 6.4 does the same for the Ctrie data structure and alternative state-of-the-art concurrent map implementations. Finally, in Section 6.5 we compare the work-stealing tree scheduling against different data-parallel schedulers.

In all the benchmarks we take the nature of a managed runtime like the JVM into account. Each benchmark is executed on a separate JVM invocation. Each method is invoked many times, and the total time is measured – this is repeated until this value stabilizes [Georges et al.(2007)Georges, Buytaert, and Eeckhout]. Effects of the JIT compiler and automatic garbage collection on performance are thus minimized. In some benchmarks we rely on the ScalaMeter benchmarking framework for Scala [Prokopec(2014b)] to ensure reproducable and stable results.

6.1 Generic Parallel Collections

Here we compare the Parallel Collections added to the Scala standard library to their sequential counterparts and other currently available parallel collections, such as Doug Lea's `extra166.ParallelArray` for Java [Lea(2014)]. All tests were executed on a 2.8 GHz 4 Dual-core AMD Opteron and a 2.66 GHz Quad-core Intel i7 (with hyper-threading). We pick the operands of our operations in a way that minimizes the amount of work done per each element. This is the worst-case for evaluating the performance of a data-parallel framework, as the abstraction and synchronization costs of processing elements cannot be amortized by the actual work.

As mentioned in Section 2.4, currently most data-parallel frameworks use concurrent combiners. Concurrent combiners are as effective as their underlying concurrent data



Total Elements: (A) 50k; (B) 100k; (C) 150k; (D) 200k

Figure 6.1: Concurrent Map Insertions

structure insert operation is scalable – a combiner has a high pressure on the `+=` method. We will compare two existing concurrent map implementations for the JVM – `ConcurrentHashMap` and `ConcurrentSkipListMap`, and then show a range of microbenchmarks comparing performance of parallel operations of various collection classes in our framework. We will see that the transformer operations using two-step combiners scale better than concurrent combiners.

In the first benchmark, a total of n elements are inserted into a concurrent collection. Insertion of these n elements is divided between p processors. This process is repeated over a sequence of 2000 runs on a single JVM invocation, and the average time required is recorded. We apply these benchmarks to `ConcurrentHashMap` and `ConcurrentSkipListMap`, two concurrent map reference implementations for the JVM, both part of the Java standard library, and compare multithreaded running time against that of inserting n elements into a `java.util.HashMap` in a single thread. The results are shown in Figure 6.1. Horizontal axis shows the number of processors and the vertical axis shows the time in milliseconds.

When a multithreaded computation consists mainly of concurrent map insertions, the performance drops due to contention. These results are not unexpected. Data structures with concurrent access are general purpose and have to support both insertions and removals, without knowing anything in advance about the data set that they will have to store. Usually they assume the 90-10 rule – for every 90 lookup operations, there are 10 operations that modify the data structure. In the case of combiners, the set of elements which comprise a collection is known and partitioned in way that allows the data structure to be populated in parallel without synchronization. The performance of `java.util.HashMap` is next compared against parallel hash tables in Figure 6.3B, in which each element of one hash table is mapped to a new hash table, and 6.3K – a similar test with the identity mapping function. Figure 6.3K shows that when no amount of computation is spent processing an element, a standard `java.util.HashMap` is faster than a parallel hash table for a single processor. As soon as the number of processors exceeds 2, the parallel hash table is constructed much faster.

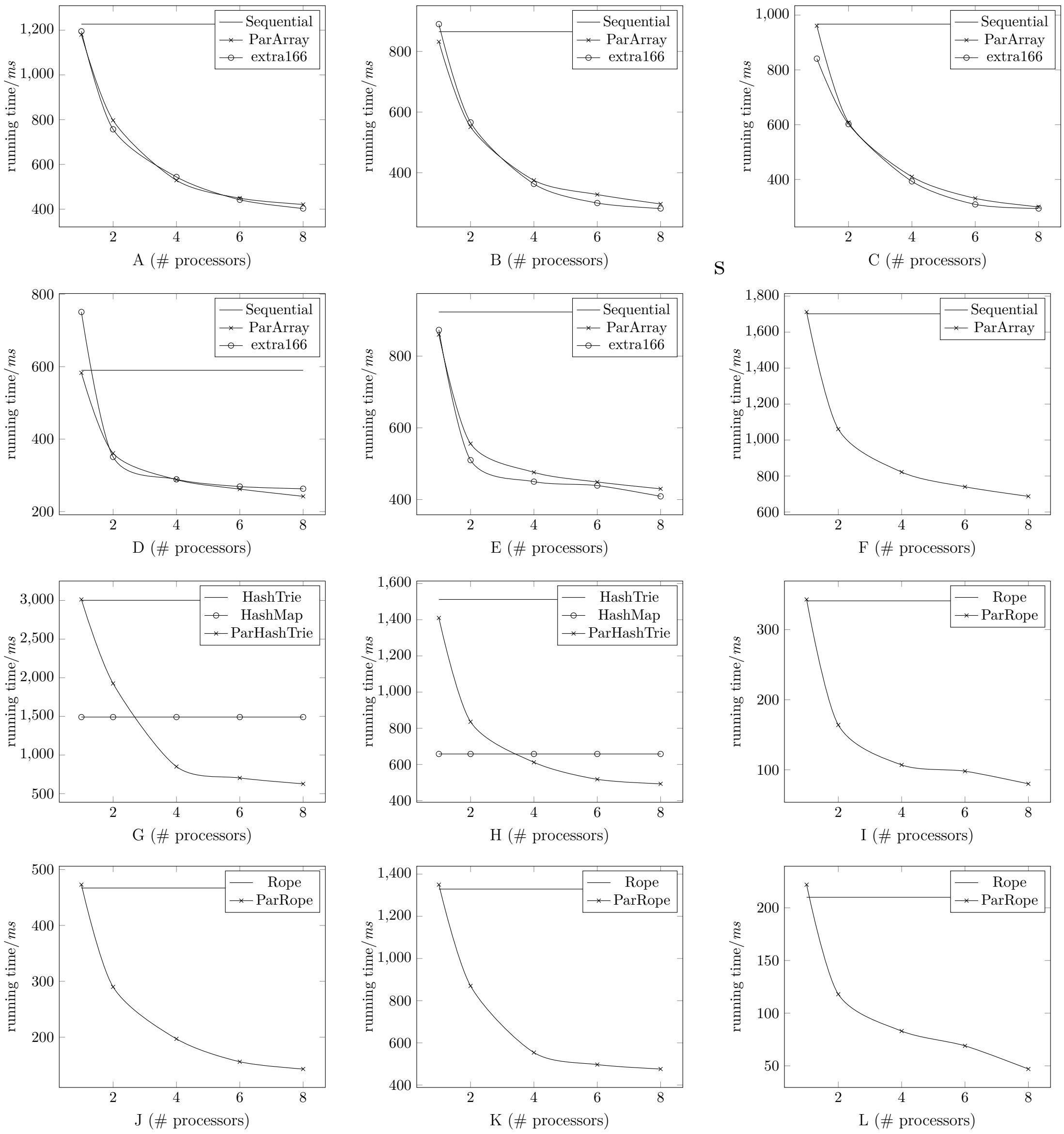
The microbenchmarks shown in Figure 6.4 were executed for parallel arrays, hash tries and hash tables on a machine with 4 Dual-Core AMD Opteron 2.8 MHz processor. The number of processors used is displayed at the horizontal axis, the time in milliseconds needed is on the vertical axis. All tests were performed for large collections, and the size of the collection is shown for each benchmark.

The entry *Sequential* denotes benchmarks for sequential loops implementing the operation in question. The entry *HashMap* denotes regular flat hash tables based on linear hashing. If the per-element amount of work is increased, data structure handling cost becomes negligible and parallel hash tries outperform hash tables even for two processors.

We should comment on the results of the `filter` benchmark. Java's parallel array first counts the number of elements satisfying the predicate, then allocates the array and copies the elements. Our parallel array assembles the results as it applies the predicate and copies the elements into the array afterwards using fast array copy operations. When using only one processor the entire array is processed at once, so the combiner contains only one chunk – no copying is required in this case, hence the reason for its high performance in that particular benchmark.

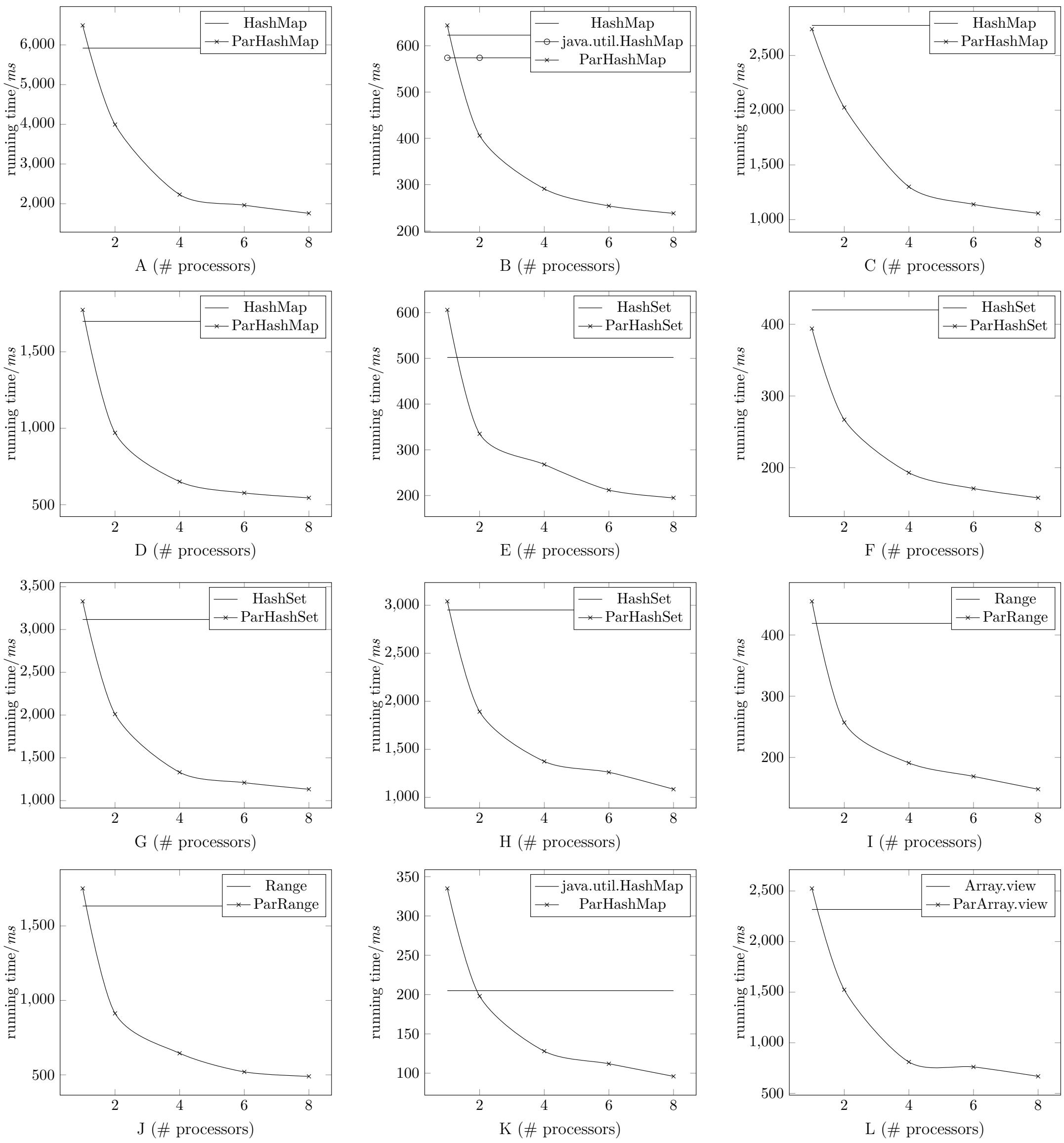
The `map` benchmark for parallel arrays uses the optimized version of the method which allocates the array and avoids copying, since the number of elements is known in advance. Benchmark for `flatMap` includes only comparison with the sequential variant, as there is currently no corresponding method in other implementations.

We show larger benchmarks in Figure 6.4. Primality testing and matrix multiplication are self-explanatory. The Coder benchmark consists of a program which does a brute-force search to find a set of all possible sentences of english words for a given sequence of digits, where each digit represents the corresponding letters on a phone key pad (e.g. '2' represents 'A', 'B' and 'C'; '43' can be decoded as 'if' or 'he'). The benchmark shown was run on a 29 digit sequence and a dictionary of around 58 thousand words. The benchmark in Figure 6.4D loads the words of the english dictionary and groups together words which have the same digit sequence.



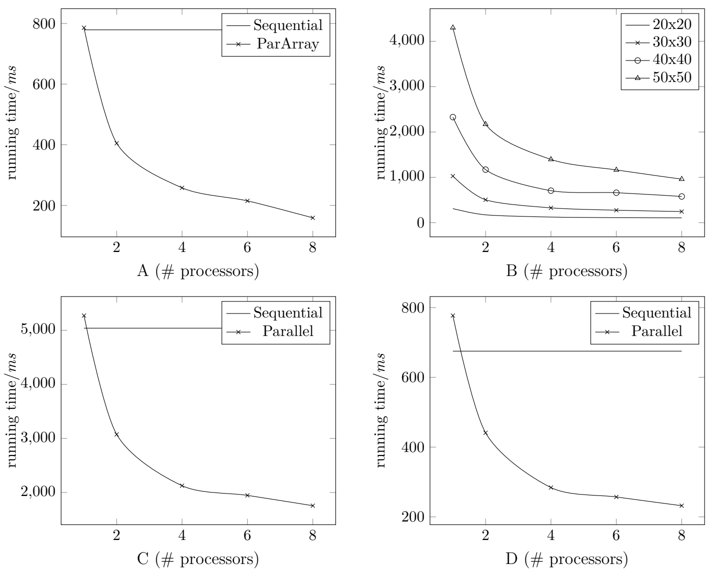
(A) `ParArray.foreach`, 200k; (B) `ParArray.reduce`, 200k; (C) `ParArray.find`, 200k; (D) `ParArray.filter`, 100k; (E) `ParArray.map`, 100k; (F) `ParArray.flatMap`, 10k; (G) `ParHashTrie.reduce`, 50k; (H) `ParHashTrie.map`, 40k; (I) `ParRope.reduce`, 50k; (J) `ParRope.map`, 50k; (K) `ParRope.filter`, 100k; (L) `ParRope.flatMap`, 100k

Figure 6.2: Parallel Collections Benchmarks I



(A) `ParHashMap.reduce`, 25k; (B) `ParHashMap.map`, 40k; (C) `ParHashMap.filter`, 40k; (D) `ParHashMap.flatMap`, 20k; (E) `ParHashSet.reduce`, 50k; (F) `ParHashSet.map`, 50k; (G) `ParHashSet.filter`, 50k; (H) `ParHashSet.flatMap`, 10k; (I) `ParRange.reduce`, 50k; (J) `ParRange.map`, 10k; (K) `ParHashMap.map(id)`, 200k; (L) `ParArray.view.reduce`, 50k

Figure 6.3: Parallel Collections Benchmarks II



(A) Primality testing; (B) Matrix multiplication; (C) Coder; (D) Grouping

Figure 6.4: Parallel Collections Benchmarks III

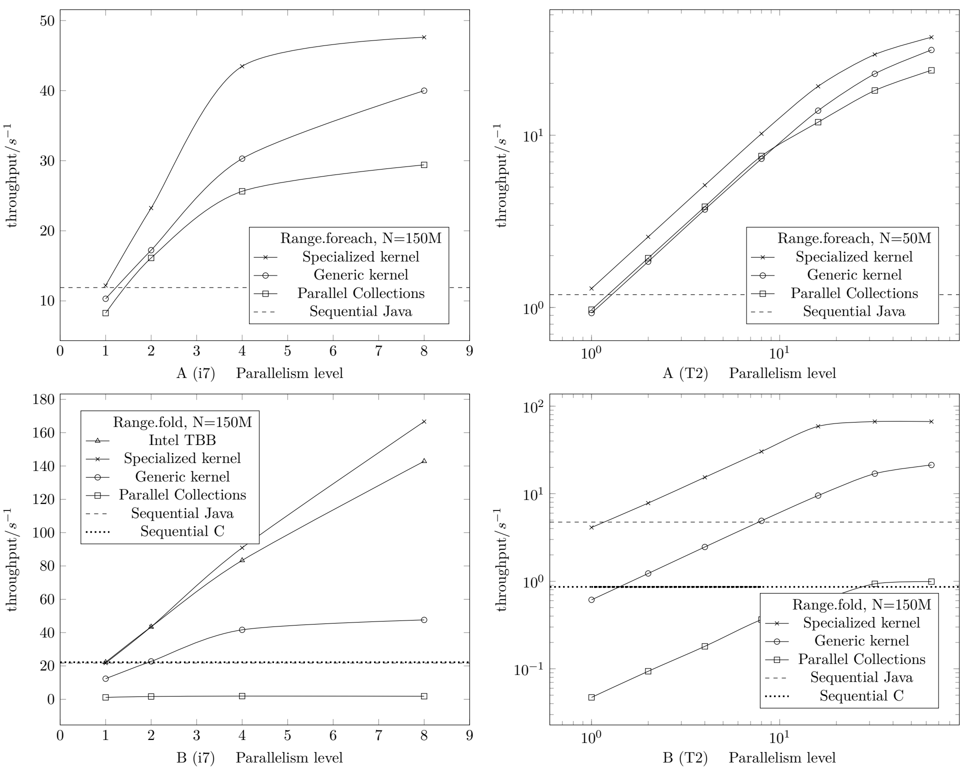
6.2 Specialized Parallel Collections

Specialized Parallel Collections in the ScalaBlitz framework [Prokopec and Petrashko(2013)] [Prokopec et al.(2014b)Prokopec, Petrashko, and Odersky] are designed to reduce abstraction to negligible levels. Scala type-specialization [Dragos and Odersky(2009)] and Scala Macros [Burmako and Odersky(2012)] are used to accomplish this, as discussed in Section 2.8. Here we identify each of the abtraction penalties separately and showing that they are overcome.

We compare against imperative sequential programs written in Java, against existing Scala Parallel Collections, a corresponding imperative C version and the Intel TBB library wherever a comparison is feasible. We show microbenchmarks addressing specific abstraction penalties on different data structures.

We perform the evaluation on the Intel i7-3930K hex-core 3.4 GHz processor with hyperthreading and an 8-core 1.2 GHz UltraSPARC T2 with 64 hardware threads. Aside from the different number of cores and processor clock, another important difference between them is in the memory throughput - i7 has a single dual-channel memory controller, while the UltraSPARC T2 has four dual-channel memory controllers. The consequence is the different scalability of these processors for write-heavy computational tasks – the scalability is dictated not only by the number of cores, but by many other factors such as the memory throughput.

The microbenchmarks in Figures 6.5 and 6.6 have a minimum cost uniform workload – the amount of computation per each element is fixed and the least possible. Those tests



Intel i7 and UltraSPARC T2; A - `ParRange.foreach`, B - `ParRange.fold`

Figure 6.5: Specialized Collections – Uniform Workload I

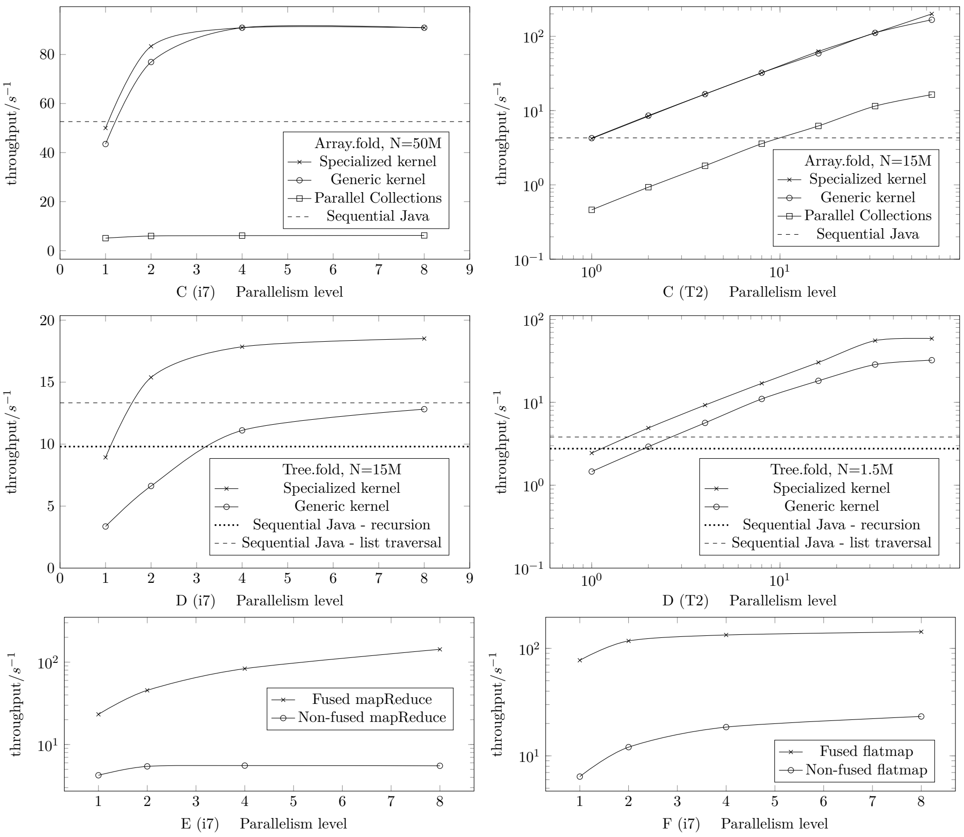
are targeted at detecting abstraction penalties discussed earlier. The microbenchmark in Figure 6.5A consists of a data-parallel `foreach` loop that occasionally sets a volatile flag (without a potential side-effect the compiler may optimize away the loop in the kernel).

```
for (i <- (0 until N).par)
  if ((i * i) & 0xffffff == 0) flag = true
```

Figure 6.5A shows a comparison between Parallel Collections, a generic work-stealing kernel and a work-stealing kernel specialized for ranges from Figure 2.21. In this benchmark Parallel Collections do not instantiate primitive types and hence do not incur the costs of boxing, but still suffer from iterator and function object abstraction penalties. Inlining the function object into the `while` loop for the generic kernel shows a considerable performance gain. Furthermore, the range-specialized kernel outperforms the generic kernel by 25% on the i7 and 15% on the UltraSPARC (note the log scale).

Figure 6.5B shows the same comparison for parallel ranges and the `fold` operation shown in the introduction:

```
(0 until N).par.fold(_ + _)
```



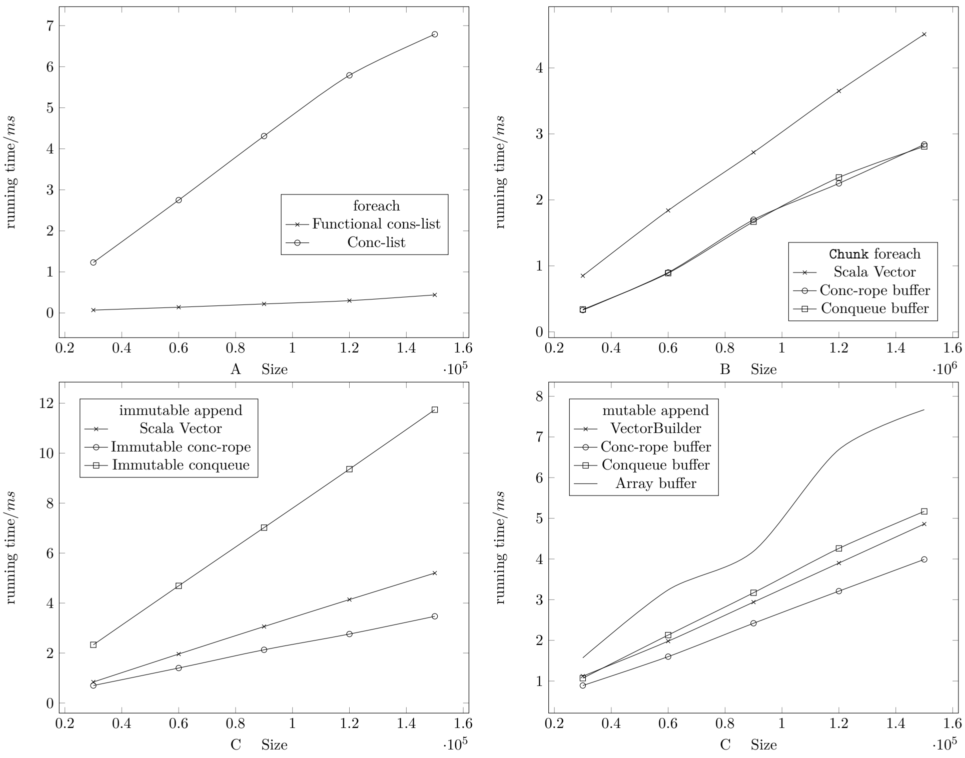
Intel i7 and UltraSPARC T2; C - `ParArray.fold`, D - `Tree.fold`, E - `mapReduce` fusion, F - `for-comprehensions` fusion

Figure 6.6: Specialized Collections – Uniform Workload II

Scala Parallel Collections performs badly in this benchmark due to abstracting over the data type, which leads to boxing. The speed gain for a range-specialized work-stealing kernel is 20× to 60× compared to Parallel Collections and 2.5× compared to the generic kernel.

Figure 6.6C shows the same `fold` microbenchmark applied to parallel arrays. While Parallel Collections again incur the costs of boxing, the generic and specialized kernel have a much more comparable performance here. Furthermore, due to the low amount of computation per element, this microbenchmark spends a considerable percentage of time fetching the data from the main memory. This is particularly noticeable on the i7 – its dual-channel memory architecture becomes a bottleneck in this microbenchmark, limiting the potential speedup to 2×. UltraSPARC, on the other hand, shows a much better scaling here due to its eight-channel memory architecture and a lower computational throughput.

The performance of the `fold` operation on balanced binary trees is shown in Figure 6.6D. Here we compare the generic and specialized `fold` kernels against a manually written recursive traversal subroutine. In the same benchmark we compare against the `fold` on



A - `foreach` w/o `Chunks`, B - `foreach` with `Chunks`, C - `append` w/o `Chunks`, D - `append` with `Chunks`

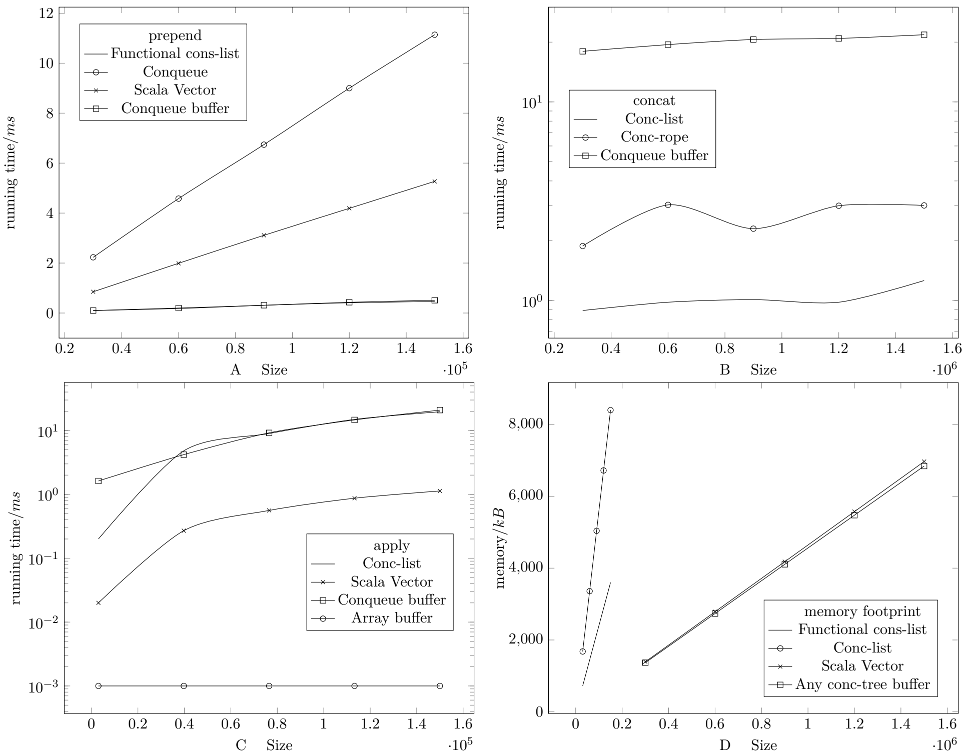
Figure 6.7: Conc-Tree Benchmarks I

functional lists from the Scala standard library commonly used in sequential functional programming. While the memory-bandwidth is the bottleneck on the i7, we again see a nice scaling on the UltraSPARC. The performance difference between the generic and the specialized kernel is 2× to 3×.

The linear scaling with respect to the sequential baseline of specialized kernels in Figures 6.5 and 6.6 indicates that the kernel approach has negligible abstraction penalties.

6.3 Conc-Trees

In this section we compare various Conc-tree variants agains fundamental sequence implementations in the Scala standard library – functional cons-lists, array buffers and Scala Vectors. The variants of these data structures exist in most other languages under different names. It is unclear if there are any functional languages that do not come with a functional cons-list implementation, and mutable linked lists are used in most general purpose languages. The defining features of a cons-list are that it prepending an element is extremely efficient, but indexing, updating or appending an elements are all $O(n)$ time operations. Functional list is an immutable data structure. Scala `ArrayBuffer` is a resizeable array implementation known as the `ArrayList` in Java and as `vector` in C++.



A - `prepend` w/o `Chunks`, B - Concatenation, C - Random Access, D - Memory Footprint
Figure 6.8: Conc-Tree Benchmarks II

Array buffers are mutable random access sequences that can index or update elements in optimal time with a simple memory read or write. Appending is also extremely efficient, but it occasionally resizes the array, having to rewrite all the elements to a new memory region. An important limitation on appending is that it takes up to 2 memory writes on average. Finally, Scala (and Clojure) Vectors are an efficient tree data structure that can implement mutable and immutable sequences. Their defining features are low memory consumption and the tree depth $O(\log_{32} n)$ bound that ensures efficient prepending and appending. Current implementations do not come with a concatenation operation. Various Conc-tree variants that we described have some or all of the above listed features.

We execute the benchmarks on an Intel i7 3.4 GHz quad-core processor with hyper-threading. We start by showing the `foreach` operation on immutable conc-lists from Section 3.1 and comparing it to the `foreach` on the functional cons-list implementation in Figure 6.7A. Traversing the cons-list is tail recursive and does not need to handle the stack state. Furthermore, inner conc-lists nodes have to be traversed even though they do not contain elements at all, which is not the case with cons-lists in which every node holds an element. The `Chunk` nodes are thus needed to ensure efficient traversal and amortize other traversal costs, as shown in Figure 6.7B. For $k = 128$ element chunks, traversal running time exceeds that of Scala Vector by almost twice for both conc-rope and conqueue buffers. In subsequent benchmarks we set k to 128.

A conc-rope buffer is essentially a two-step array combiner, and the `foreach` benchmark validates the fact that conc-ropes are a better choice for data-parallelism from the traversal point of view than Vectors. Thus, we next evaluate the append performance. We start with the immutable Conc-trees that do not use `Chunk` nodes in Figure 6.7C. While more housekeeping causes conqueues to be twice as slow compared to immutable Vectors, immutable conc-ropes are more efficient. This is not the case with mutable conqueues in Figure 6.7D, where we compare conc-rope and conqueue buffers against mutable Vector version called a `VectorBuilder`. Again, conc-ropes are clearly the data structure of choice for sequence combiners. All three of them outperform array buffers, since array buffers require more than one write to an array per element due to resizing.

When it comes to just prepending elements, functional cons-list is very efficient – prepending amounts to creating a single node. Scala functional list have the same performance as mutable conqueue buffers, even though they are themselves immutable data structures. Both Scala Vectors and immutable conqueues are an order of magnitude slower.

Concatenation has the same performance for both immutable and mutable Conc-tree variants[1] – we show it in Figure 6.8B, where we repeat concatenation 10^4 times. Concatenating conc-ropes is slightly more expensive than conc-list concatenation because of the normalization, and it varies with size because the number of trees (that is, non-zeros) in the append list fluctuates. Conqueue concatenation is an order of magnitude slower (note the log axis) due to the longer normalization process[2]. Concatenating lists, array buffers and Scala Vectors is not shown here as it is thousands of times slower for these data structure sizes.

Random access to elements is an operation where Scala Vectors have a clear upper hand over the other immutable sequence data structures. Although indexing Scala Vector is an order of magnitude faster than indexing Conc-trees, both are orders of magnitudes slower than random access in arrays. In defense of Conc-trees, applications that really care about performance would convert immutable sequences into arrays before carrying out many indexing operations, and would not use `Vector` in the first place.

Finally, we show memory consumption in Figure 6.8D. While a conc-list occupies approximately twice as much memory as a functional cons-list, the choice of using `Chunk` nodes has a clear impact on the memory footprint – both Scala Vectors and Conc-trees with `Chunk` nodes occupy an almost optimal amount of memory, where optimal is the number of elements in the data structure times the pointer size.

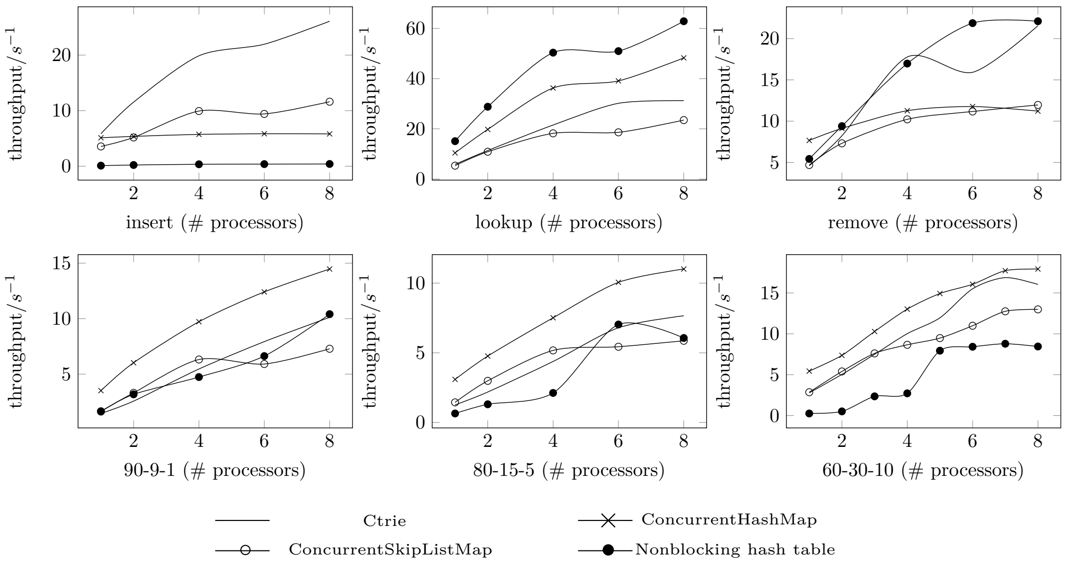


Figure 6.9: Basic Ctrie Operations, Quad-core i7

6.4 Ctries

We performed experimental measurements for Ctrie operations on three separate architectures – a JDK6 configuration with a quad-core 2.67 GHz Intel i7 processor with 8 hyperthreads, a JDK6 configuration with an 8-core 1.165 GHz Sun UltraSPARC-T2 processor with 64 hyperthreads and a JDK7 configuration with four 8-core Intel Xeon 2.27 GHz processors with a total of 64 hyperthreads. The first configuration has a single multicore processor, the second has a single multicore processor, but a different architecture and the third has several multicore processors on one motherboard. We followed established performance measurement methodologies [Georges et al.(2007)Georges, Buytaert, and Eeckhout]. We compared the performance of the Ctrie against the `ConcurrentHashMap` and the `ConcurrentSkipListMap` from the Java standard library, as well as the Cliff Click's non-blocking concurrent hash map implementation [Click(2007)]. All of the benchmarks show the number of threads used on the x-axis and the throughput on the y-axis. In all experiments, the Ctrie supports the snapshot operation.

The first benchmark called *insert* starts with an empty data structure and inserts $N = 1000000$ entries into the data structure. The work of inserting the elements is divided equally between P threads, where P varies between 1 and the maximum number of hyperthreads on the configuration (x-axis). The y-axis shows throughput – the number of times the benchmark is repeated per second. This benchmark is designed to test the

[1] Mutable variants require taking a snapshot. The basic Conc-tree is an immutable data structure with efficient modification operations, so snapshot can be done lazily as in Section 4.2.2.

[2] Despite the negative ring to a phrase *a magnitude slower*, that is still very, very fast.

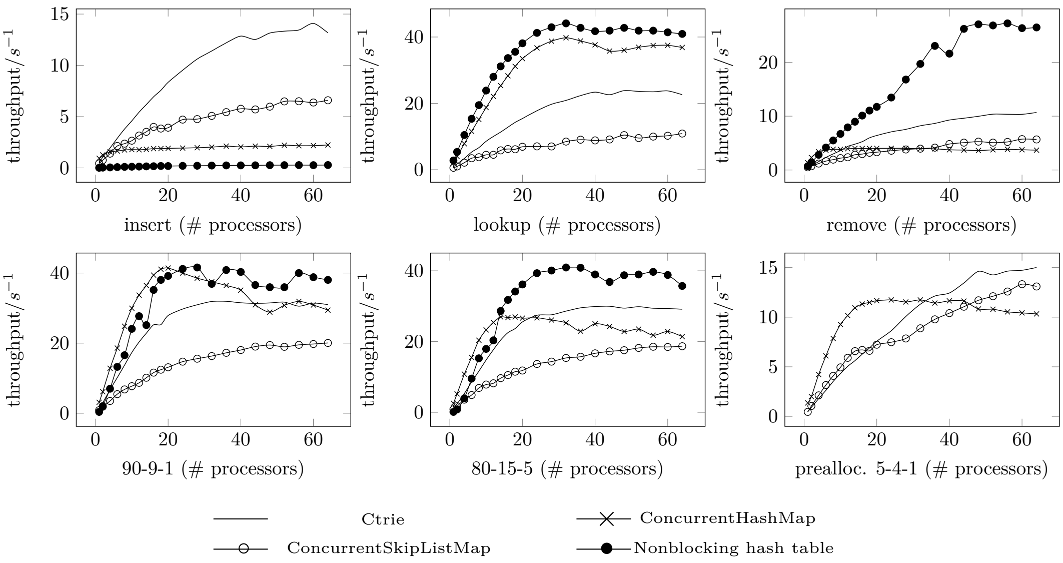


Figure 6.10: Basic Ctrie Operations, 64 Hardware Thread UltraSPARC-T2

scalability of the resizing, since the data structure is initially empty. Data structures like hash tables, which have a resize phase, do no seem to be very scalable for this particular use-case, as shown in Figure 6.9. On the Sun UltraSPARC-T2 (Figure 6.10), the Java concurrent hash map scales for up to 4 threads. Cliff Click's nonblocking hash table scales, but the cost of the resize is so high that this is not visible on the graph. Concurrent skip lists scale well in this test, but Ctries are a clear winner here since they achieve an almost linear speedup for up to 32 threads and an additional speedup as the number of threads reaches 64.

The benchmark *lookup* does $N = 1000000$ lookups on a previously created data structure with N elements. The work of looking up all the elements is divided between P threads, where P varies as before. Concurrent hash tables perform especially well in this benchmark on all three configurations – the lookup operation mostly amounts to an array read, so the hash tables are $2 - 3$ times faster than Ctries. Ctries, in turn, are faster than skip lists due to a lower number of indirections, resulting in fewer cache misses.

The *remove* benchmark starts with a previously created data structure with $N = 1000000$ elements. It removes all of the elements from the data structure. The work of removing all the elements is divided between P threads, where P varies. On the quad-core processor (Figure 6.9) both the Java concurrent skip list and the concurrent hash table scale, but not as fast as Ctries or the Cliff Click's nonblocking hash table. On the UltraSPARC-T2 configuration (Figure 6.10), the nonblocking hash table is even up to 2.5 times faster than Ctries. However, we should point out that the nonblocking hash table does not perform compression – once the underlying table is resized to a certain size, the memory is used regardless of whether the elements are removed. This can be a problem for long running

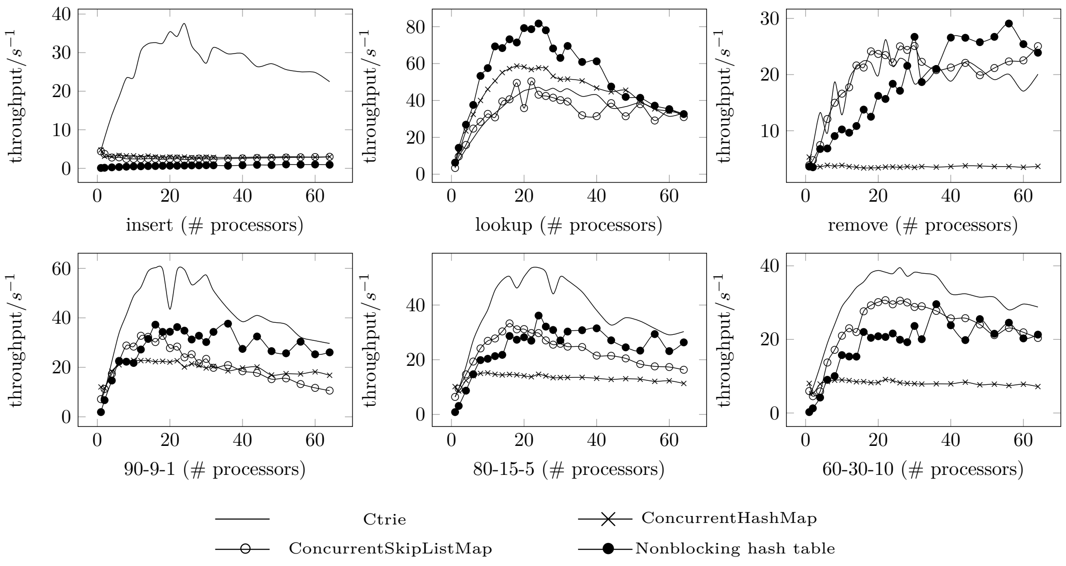


Figure 6.11: Basic Ctrie Operations, 4x 8-core i7

applications and applications using a greater number of concurrent data structures.

The next three benchmarks called $90 - 9 - 1$, $80 - 15 - 5$ and $60 - 30 - 10$ show the performance of the data structures when the operations are invoked in the respective ratio. Starting from an empty data structure, a total of $N = 1000000$ invocations are done. The work is divided equally among P threads. For the $90 - 9 - 1$ ratio the Java concurrent hash table works very well on both the quad-core configuration and the UltraSPARC-T2. For the $60 - 30 - 10$ ratio Ctries seem to do as well as the nonblocking hash table. Interestingly, Ctries seem to outperform the other data structures in all three tests on the 4x 8-core i7 (Figure 6.11).

The $preallocated - 5 - 4 - 1$ benchmark in Figure 6.10 proceeds exactly as the previous three benchmarks with the difference that it starts with a data structure that contains all the elements. The consequence is that the hash tables do not have to be resized – this is why the Java concurrent hash table performs better for P up to 16, but suffers a performance degradation for bigger P. For $P > 32$ Ctries seem to do better. In this benchmarks, the nonblocking hash table was 3 times faster than the other data structures, so it was excluded from the graph. For applications where the data structure size is known in advance this may be an ideal solution – for others, preallocating may result in a waste of memory.

To evaluate snapshot performance, we do 2 kinds of benchmarks. The $snapshot - remove$ benchmark in Figure 6.12 is similar to the *remove* benchmark – it measures the performance of removing all the elements from a snapshot of a Ctrie and compares that time to removing all the elements from an ordinary Ctrie. On both i7 configurations (Figures 6.9 and 6.11), removing from a snapshot is up to 50% slower, but scales in the

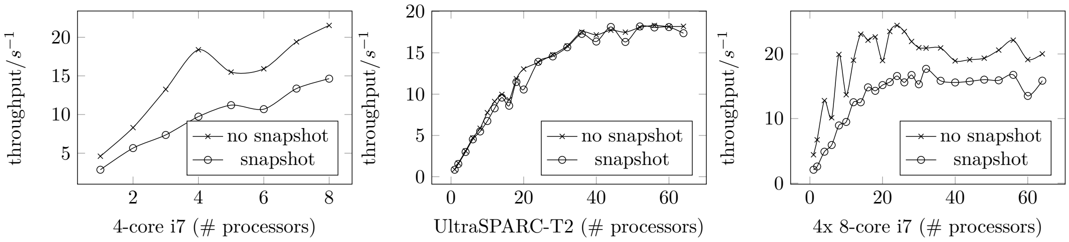


Figure 6.12: Remove vs. Snapshot Remove

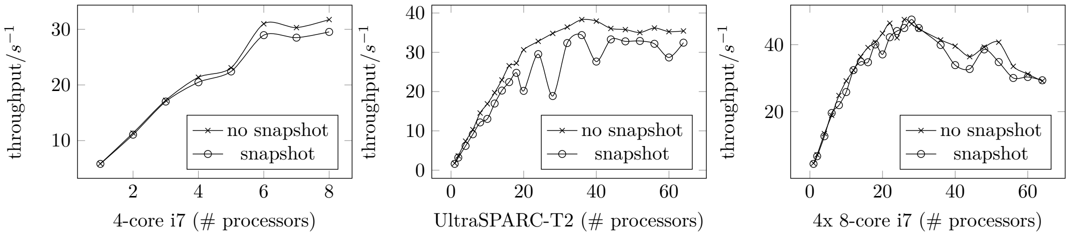


Figure 6.13: Lookup vs. Snapshot Lookup

same way as removing from an ordinary Ctrie. On the UltraSPARC-T2 configuration (Figure 6.10), this gap is much smaller. The benchmark *snapshot − lookup* in Figure 6.13 is similar to the last one, with the difference that all the elements are looked up once instead of being removed. Looking up elements in the snapshot is slower, since the Ctrie needs to be fully reevaluated. Here, the gap is somewhat greater on the UltraSPARC-T2 configuration and smaller on the i7 configurations.

Finally, the *PageRank* benchmark in Figure 6.14 compares the performance of iterating parts of the snapshot in parallel against the performance of filtering out the page set in each iteration as explained in Section 4.2.2. The snapshot-based implementation is much faster on the i7 configurations, whereas the difference is not that much pronounced on the UltraSPARC-T2.

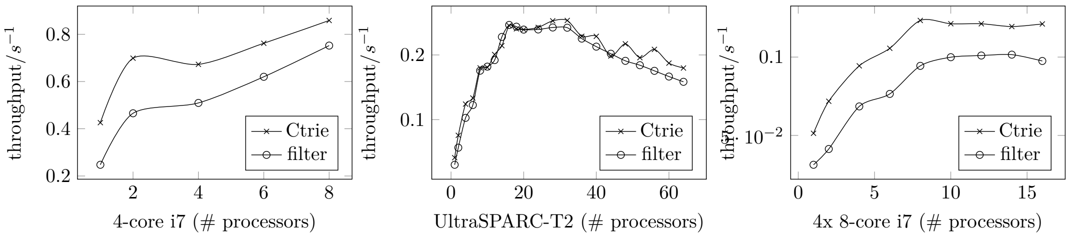
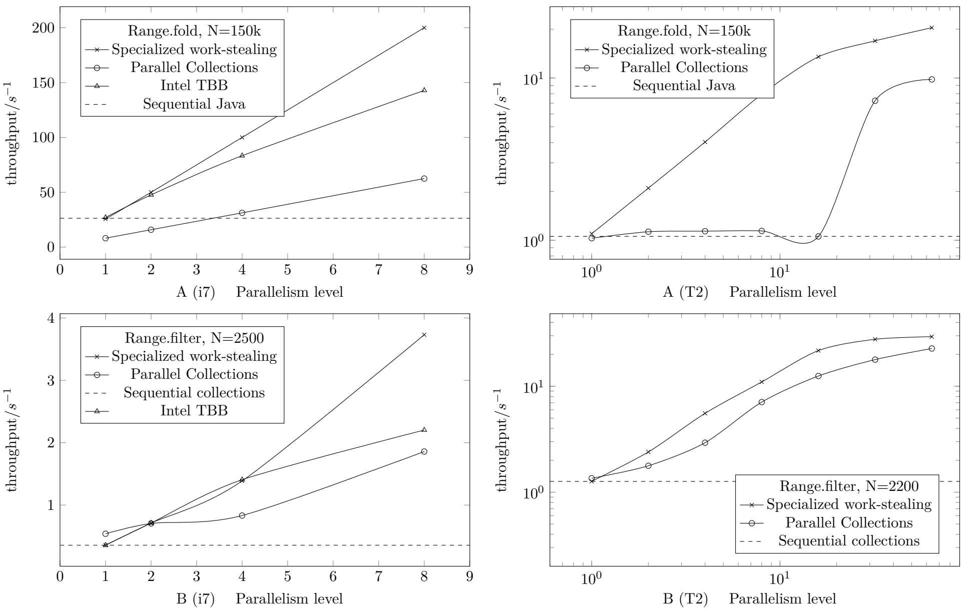


Figure 6.14: PageRank with Ctries



Intel i7 and UltraSPARC T2; A - *step* workload, B - *exponential* workload
Figure 6.15: Irregular Workload Microbenchmarks

6.5 Work-stealing Tree Data-Parallel Scheduling

We have already compared the work-stealing tree scheduler against several other data-parallel schedulers in Section 5.2.5, in Figures 5.13 and 5.14. In this section we focus on comparisons with the Scala Parallel Collections scheduler from Section 2.7 and Intel TBB [Reinders(2007)].

Figure 6.15 shows improved performance compared to Parallel Collections not only due to overcoming abstraction penalties, but also due to improved scheduling. The `Splitter` abstraction divides an iterator into its subsets *before* the parallel traversal begins. The scheduler chooses a batching schedule for each worker such that the batch sizes increase exponentially. As discussed in Section 2.7, this scheduler is adaptive – when a worker steals a batch it divides it again. However, due to scheduling penalties of creating splitters and task objects, and then submitting them to a thread pool, this subdivision only proceeds up to a fixed threshold $\frac{N}{8P}$, where N is the number of elements and P is the number of processors. Concentrating the workload in a sequence of elements smaller than the threshold yields a suboptimal speedup. Work-stealing iterators, on the other hand, have much smaller batches with a potential single element granularity.

In Figure 6.15A we run a parallel `fold` method on a *step* workload $\chi(0.97, \frac{n}{N})$ – the first 97% of elements have little associated with them, while the rest of the elements require a high amount of computation. Intel TBB exhibits a sublinear scaling in this microbenchmark, being about 25% slower compared to the work-stealing tree scheduling. Due to a predetermined work scheduling scheme where the minimum allowed batch size

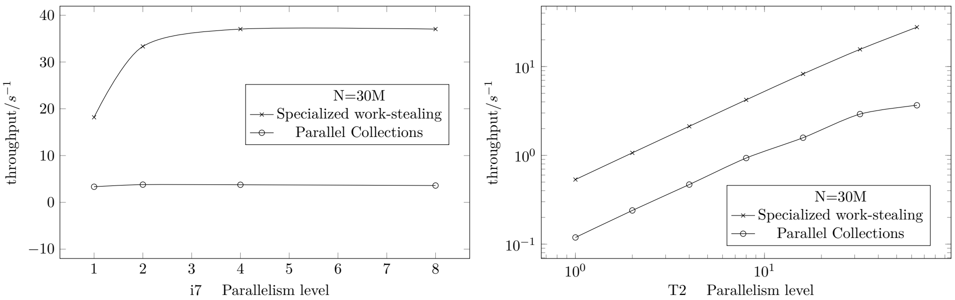


Figure 6.16: Standard Deviation on Intel i7 and UltraSPARC T2

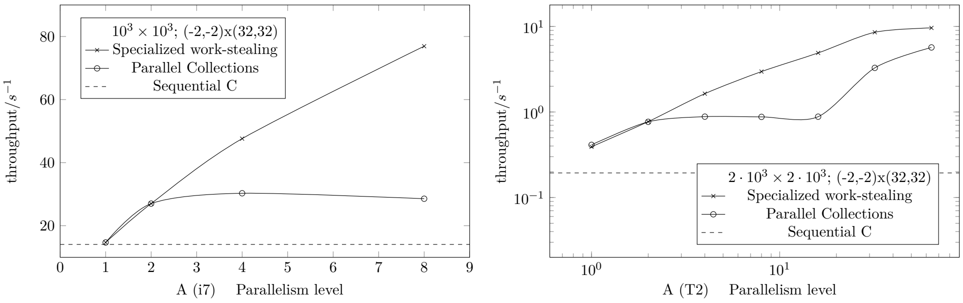


Figure 6.17: Mandelbrot Set Computation on Intel i7 and UltraSPARC T2

depends on the number of threads, the task-based scheduler only yields a speedup on UltraSPARC with more than 16 threads.

As shown in Figure 6.15B, Intel TBB is up to 2× slower compared to work-stealing tree scheduling for an *exponential* workload where the amount of work assigned to the n-th element grows with the function $2^{\frac{n}{100}}$, while the existing Scala Parallel Collections do not cope with it well. In Figure 6.16 we show performance results for an application computing a standard deviation of a set of measurements. The relevant part of it is as follows:

```
val mean = measurements.sum / measurements.size
val variance = measurements.aggregate(0.0)(_ + _) {
  (acc, x) => acc + (x - mean) * (x - mean)
}
```

As in the previous experiments, Parallel Collections scale but have a large constant penalty due to boxing. On UltraSPARC boxing additionally causes excessive memory accesses resulting in non-linear speedups for higher parallelism levels ($P = 32$ and $P = 64$).

To show that these microbenchmarks are not just contrived examples, we show several larger benchmark applications as well. We first show an application that renders an image of the Mandelbrot set in parallel. The Mandelbrot set is irregular in the sense that all points outside the circle $x^2 + y^2 = 4$ are not in the set, but all the points within the circle require some amount of computation to determine their set membership. Rendering an image a part of which contains the described circle thus results in an irregular workload.

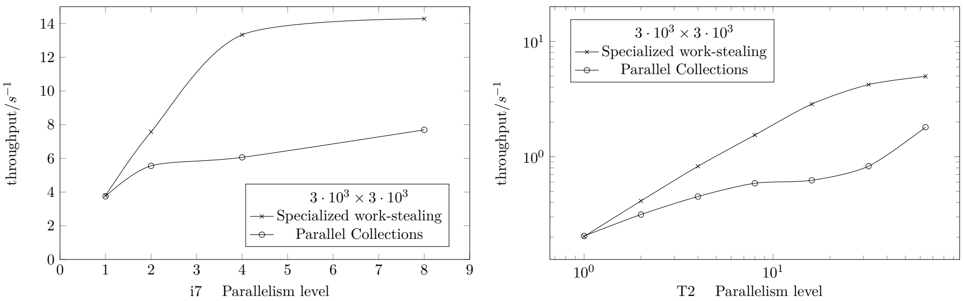


Figure 6.18: Raytracing on Intel i7 and UltraSPARC T2

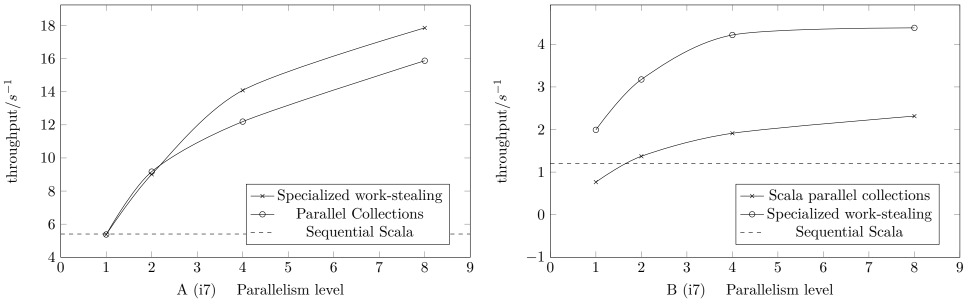


Figure 6.19: (A) Barnes-Hut Simulation; (B) PageRank on Intel i7

We show the running times of rendering a Mandelbrot set in Figure 6.17. In Figure 6.17 the aforementioned computationally demanding circle is in the lower left part of the image. We can see a similar effect as in the Figure 6.15A – with a fixed threshold there is only a 50% to 2× speedup until P becomes larger than 16.

In Figure 6.18 we show the performance of a parallel raytracer, implemented using existing Parallel Collections and work-stealing tree scheduling. Raytracing renderers project a ray from each pixel of the image being rendered, and compute the intersection between the ray and the objects in the scene. The ray is then reflected several times up until a certain threshold. This application is inherently data-parallel – computation can proceed independently for different pixels. The workload characteristics depend on the placement of the objects in the scene. If the objects are distributed uniformly throughout the scene, the workload will be uniform. The particular scene we choose contains a large number of objects concentrated in one part of the image, making the workload highly irregular.

The fixed threshold on the batch sizes causes the region of the image containing most of the objects to end up in a single batch, thus eliminating most of the potential parallelism. On the i7 Parallel Collections barely manage to achieve the speedup of 2×, while the data structure aware work-stealing easily achieves up to 4× speedups. For higher parallelism levels the batch size becomes small enough to divide the computationally expensive part of the image between processors, so the plateau ends at $P = 32$ on UltraSPARC. The speedup gap still exists at $P = 64$ – existing Parallel Collections scheduler is 3× slower than the work-stealer tree scheduler.

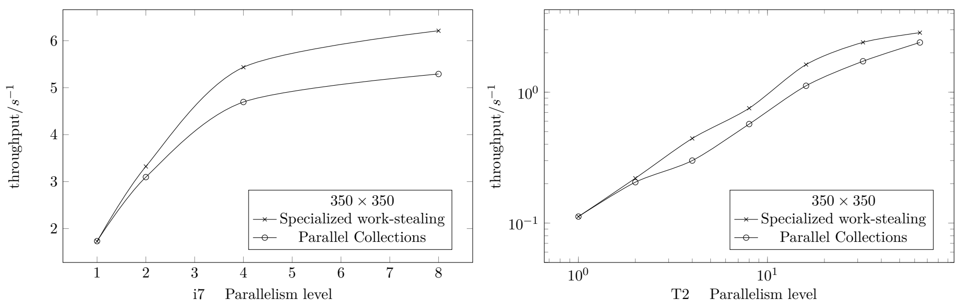


Figure 6.20: Triangular Matrix Multiplication on Intel i7 and UltraSPARC T2

We have parallelized the Barnes-Hut n-body simulation algorithm. This simulation starts by finding the bounding box of all the particles, and then dividing them into a fixed number of rectangular sectors within that bounding box, both in parallel. Quadtrees are then constructed in parallel for the particles within each sector and merged into a singular quadtree. Finally, the positions and speeds of all the particles are updated in parallel using the quad tree to approximate the net force from the distant particles.

We used the Barnes-Hut simulation to simulate the movement of two stellar bodies composed of $25k$ stars. We recorded the average simulation step length across a number of simulation iterations. Although this turned out to be a relatively uniformly distributed workload with most of the stars situated in the sectors in the middle of the scene, we still observed a consistent 10% increase in speed with respect to preemptive scheduling in Scala Parallel Collections, as shown in Figure 6.19A.

The PageRank benchmark in Figure 6.19B shows how fusing the operations such as `map`, `reduce`, `groupBy` and `aggregate` leads to significant speedups. The single-threaded work-stealing tree version is already 35% faster compared to the standard sequential Scala collections.

The last application we choose is triangular matrix multiplication, in which a triangular $n \times n$ matrix is multiplied with a vector of size n. Both the matrix and the vector contain arbitrary precision values. This is a mild irregular workload, since the amount of work required to compute the i-th element in the resulting vector is $w(i) = i$ – in Figure 5.13 we called this workload triangular.

Figure 6.20 shows a comparison of the existing Parallel Collections scheduler and data-structure-aware work-stealing. The performance gap is smaller but still exists, work-stealing tree being 18% faster on the i7 and 20% faster on the UltraSPARC. The downsides of fixed size threshold and preemptive batching are thus noticeable even for less irregular workloads, although less pronounced.

7 Conclusion

This thesis presented a data-parallel programming framework for shared-memory multiprocessors. The data-parallel computing model was applied to single-threaded data structures operating under the bulk-synchronous processing and quiescence assumptions, and to concurrent data structures that allow concurrent modifications. We have shown how to efficiently assign data-parallel workloads to different processors, and how to handle particularly irregular workloads. We hope that the readers had as much fun reading it as we had writing it.

We feel that the research done as part of this thesis had important impact on both the state-of-the-art of concurrent programming and the ecosystem of several programming languages. The basic data-parallel framework described in Chapter 2 is now a part of the Scala standard library Parallel Collections. An abstraction called *spliterator* similar to our splitter is now a part of JDK8. The Ctrie data structure from Chapter 4 was originally implemented in Scala and subsequently made part of the Scala standard library. It was already independently reimplemented in other languages like Java [Levenstein(2012)] and Common Lisp [Lentz(2013)], and is available as a Haskell module [Schröder(2014)]. A monotonically growing variant of the Ctrie is used in Prolog for tabling techniques [Areias and Rocha(2014)]. The work-stealing tree scheduler is now a part of the efficient data-parallel computing module written in Scala called ScalaBlitz [Prokopec and Petrashko(2013)].

The concurrent data structure parallelization through the use of efficient, linearizable, lock-free snapshots presented in this thesis poses a significant breakthrough in the field of concurrent data structures. We have shown how to apply the snapshot approach to two separate lock-free concurrent data structures and identified the key steps in doing so A plethora of interesting lock-free concurrent data structures amenable to the snapshot approach remain unexplored – we feel confident that the findings in this thesis will drive future research in this direction.

A Conc-Tree Proofs

This appendix does not aim to prove the correctness of the Conc-tree data structure. The basic Conc-tree data structure is used by a single thread, and reasoning about its operations is intuitive – the Conc-tree correctness is much easier to establish than that of lock-free concurrent data structures shown later. Thus, we do not consider it valuable to formalize the effects of the Conc-tree operations. Instead, we focus our efforts towards proving the claims about the asymptotic running time bounds from Chapter 3. We assume that the reader is well acquainted with the content of the Chapter 3.

Throughout the appendix, we use the terms *height*, *depth* and *level* interchangeably, and mean – the longest path from the root to some leaf. When we say *rank*, we refer to the depth of the Conc-trees in the `Num` node of the conqueue.

We start by recalling the basic theorems about the performance of the concatenation operation on Conc-tree list. Some of these theorems were already proved earlier – we refer the reader to the appropriate section where necessary.

Theorem A.1 (Conc-tree Height) *A Conc-tree list with n elements has $O(\log n)$ depth.*

Proof. These bounds follow directly from the Conc-tree list invariants, and were proved in Section 3.1. □

The following corollary is a direct consequence of this theorem, and the fact that the Conc-tree list lookup operation traverses at most one path from the root to a leaf:

Corollary A.2 *The Conc-tree list lookup operation (*`apply`*) runs in $O(\log n)$ time, were n is the number of elements contained in the Conc-tree list.*

Lemma A.3 (Linking Cost) *Conc-tree linking (`new <>`) is an $O(1)$ operation.*

Proof. Here, we assume that a memory allocator allocates an object in worst-case constant time. Proving the lemma is then trivial. We note that even without the fineprint about the memory allocator, the linking cost is in practice acceptable, and the approximation of this assumption is reasonable – it is the goal of most memory allocators to allocate small objects in amortized constant time with very low constant factors. □

Theorem A.1 also dictates the running time of the `update` operation. Recall that the `update` operation does not insert a new node into the Conc-tree list, but replaces a specific leaf, and re-links the trees on the path between the root and the leaf. The amount of work done during the update is bound by the depth of the Conc-tree list and the cost of the linking operation from Lemma A.3.

Corollary A.4 *The Conc-tree list update operation (`update`) runs in $O(\log n)$ time, were n is the number of elements contained in the Conc-tree list.*

Showing the asymptotic running time of the insert operation (`insert`) is slightly more demanding, as it involves Conc-tree list concatenation, rather than just simple Conc-tree linking. We first consider several fundamental theorems about Conc-trees.

Theorem A.5 (Height Increase) *Concatenating two Conc-tree lists of heights h_1 and h_2 yields a tree with height h such that $h - \max(h_1, h_2) \leq 1$.*

Proof. This theorem was proved in Section 3.1, by analyzing the cases in the `concat` method. □

Theorem A.6 (Concatenation Running Time) *Concatenating two Conc-tree lists of heights h_1 and h_2 is an $O(|h_1 - h_2|)$ asymptotic running time operation.*

Proof. This theorem was proved in Section 3.1. □

The following corollary is a simple consequence of the fact that in $O(n)$ time, a program can visit (e.g. allocate) only $O(n)$ space.

Corollary A.7 *Concatenating two Conc-tree lists of heights h_1 and h_2, respectively, allocates $O(|h_1 - h_2|)$ nodes.*

Note that the `insert` method was particularly picky about the order in which it zips the trees after adding the node. It turns out that this order drives the running time of the `insert` method, and must be done bottom-up.

Theorem A.8 (Increasing Concatenation Sequence) *Take any sequence of Conc-tree lists $c_0, c_1, c_2, \dots, c_k$ of increasing heights $h_0, h_1, h_2, \dots, h_k$, such that no two trees have the same height, except possibly the first two, whose height differs by* 1*:*

$$h_0 - 1 \leq h_1 < h_2 < \dots < h_k$$

Let C denote the left-to-right concatenation of the Conc-tree lists $c_0, c_1, c_2, \dots, c_k$:

$$C = (\dots((c_0 \diamond c_1) \diamond c_2) \diamond \dots) \diamond c_k)$$

Computing the Conc-tree list C takes $O(h_k)$ time, where h_k is the height of the largest Conc-tree c_k.

Proof. We prove this by induction. For a sequence of one or two Conc-tree lists, the claim trivially follows from the Theorem A.6. Now, assume that we need to concatenate a sequence of Conc-tree lists $C_l \leq c_{l+1} < c_{l+2} < \dots < c_k$ with heights:

$$H_l - 1 \leq h_{l+1} < h_{l+2} < \dots < h_k$$

The Conc-tree list C_l is the concatenation accumulation of the first $l+1$ Conc-tree lists. From the induction hypothesis, the total amount of work to compute C_l is bound by $O(h_l)$, where h_l is the height of the largest Conc-tree list c_l previously merged to obtain C_l. By Theorem A.6, concatenating C_l and c_{l+1} runs in $O(h_{l+1} - H_l)$ time, making the total amount of work bound by $O(h_l + h_{l+1} - H_l)$. From Theorem A.5 and the inductive hypothesis that $H_{l-1} - 1 \leq h_l$, we know that $H_l - h_l \leq 2$. It follows the total amount of work performed so far is bound by $O(h_{l+1})$. Also, by Theorem A.5, concatenating C_l and c_{l+1} yields a Conc-tree list C_{l+1}, whose height is $H_{l+1} \leq H_l + 1 \leq h_{l+1} + 2 \leq h_{l+2} + 1$. We end up with a sequence of Conc-tree lists $C_{l+1} \leq c_{l+2} < c_{l+3} < \dots < c_k$ with heights:

$$H_{l+1} - 1 \leq h_{l+2} < h_{l+3} < \dots < h_k$$

which proves the inductive step – eventually, we concatenate all the trees in $O(h_k)$ time, and arrive at the Conc-tree list C_k. Note that $C_k = C$. □

Lemma A.9 (Rope Spine Length) *A Conc-tree rope can contain at most $O(\log n)$* `Append` *nodes, where n is the number of elements in the Conc-tree rope.*

Proof. Consider a strictly increasing sequence R of Conc-tree lists, containing in total n elements. Such a sequence has two extreme cases. The first is that R contains a single

Conc-tree list, which contains all n elements. In this case, a Conc-tree rope append operation takes $O(1)$ time by Lemma A.3 – appending a single-element Conc-tree list is a simple linking operation. The other extreme case is that the sequence R contains k Conc-tree lists, heights of which differ by exactly 1. In this case:

$$\frac{2}{5-\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{k+1}-1) = \sum_{i=0}^{k} \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{i} \approx \sum_{i=0}^{k} F(i) \leq n \leq \sum_{i=0}^{k} 2^{i} = 2^{k+1}-1$$

where $F(i)$ is the i-th Fibonacci number. It follows that k is bound by $\Theta(\log n)$. At any time, there can be at most $O(\log n)$ Conc-tree lists in the sequence R. □

In the Conc-tree list split operation, and the Conc-tree rope normalization, series of concatenations proceed in strictly increasing manner. In the the Conc-tree list insert operation, series of concatenations proceed in a similar manner, but the first Conc-tree in the series of concatenation can always be higher by at most 1. The following corollaries are a consequence of the Theorem A.8 and the Lemma A.9.

Corollary A.10 *The Conc-tree insert operation (*`insert`*), Conc-tree list split operation (*`split`*), and Conc-tree rope normalization (*`normalize`*) run in $O(\log n)$ time, were n is the number of elements contained in the Conc-tree list.*

Note that the normalization bound affects the basic Conc-tree rope operations. The only difference with respect to the Conc-tree lists is the jump to $O(\log \max(n_1, n_2))$ time Conc-tree rope concatenation from the $O(\log \frac{n_1}{n_2})$ Conc-tree list concatenation.

Corollary A.11 *Conc-tree rope concatenation (*`concat`*) runs in $O(\log n)$ time.*

Lemma A.9 has a direct consequence on the running time of the persistent append operation, as the persistent use of the Conc-tree rope can maintain an instance in the worst-case configuration with the maximum number of `Append` nodes.

Corollary A.12 (Persistent Append Running Time) *The persistent Conc-tree rope append operation has worst-case $O(\log n)$ running time.*

An ephemeral Conc-tree rope append operation benefits from amortization, and has an improved running time bound.

Theorem A.13 (Ephemeral Append Running Time) *The ephemeral Conc-tree rope append operation has amortized $O(1)$ running time.*

Proof. First, note that a sequence of append operations in a Conc-tree rope corresponds to incrementing binary numbers, where a Conc-tree list at level l in the spine corresponds to a binary 1-digit at position l in the binary number, the lack of a Conc-tree list at level l corresponds to a binary 0-digit at position l in the binary number, and linking two Conc-tree lists of height l corresponds to a carry operation. We need to prove that, given that the cost of a carry operation is $O(1)$, the average amount of work in n binary number increments approaches $O(1)$, as n grows.

To do this, we consider a binary number with w digits, in which the digit at the highest position $w - 1$ has just become 1. After $O(2^w)$ increment operations, the digit at the position $w - 1$ again becomes 0. We count how many digits were flipped from 1 to 0 during these $O(2^w)$ increment operations, in other words – we count the carry operations. The digit at position $w - i$ is flipped $O(2^i)$ times, so the total flip count is $O(2^w)$. It follows that the average amount of work is $O(1)$. □

Having proved the running times of basic Conc-tree list and Conc-tree rope operations, we turn to conqueues.

Theorem A.14 (Tree shaking) *The left-shake (right-shake) operation always returns a tree that is either left-leaning or locally balanced (respectively, right-leaning or locally balanced). If the height of the resulting tree is different than the input tree, then the resulting tree is locally balanced. The resulting tree is by at most* 1 *lower than the input tree.*

Proof. Trivially, by analyzing the cases in the shake-left operation (`shakeLeft`), and the analogous shake-right operation (`shakeRight`). □

Going from amortized time to worst-case time is typically achieved by employing some sort of *execution scheduling* [Okasaki(1996)]. In this technique, parts of the work are performed incrementally, rather than letting work accumulate and eventually *cascade* throughout the data structure, distancing the worst-case bound from the amortized one. Analysis is made more difficult by the fact that we need to prove not only that the total amount of work and the total amount of operations have the same asymptotic complexity, but also need to show the exact ratio between the two. In worst-case analysis of persistent data structures that rely on execution scheduling, constant factors matter not only for performance, but also for proving the exact running time bounds.

Lazy conqueues rely on a variant of execution scheduling to avoid cascading. The importance of a precise ratio between the work spent in the operation and the work spent scheduling is apparent for conqueues. As we will see, the conqueue `pushHead` operation (which corresponds to an increment operation in a positional 4-digit base-2 number system) requires evaluating a single Spine node (corresponding to a single carry operation). By contrast, the popHead operation (which corresponds to a decrement operation in the same number system) requires evaluating two Spine nodes (corresponding to two borrow operations). This extra work in the borrow case results from the fact that the Conc-tree lists are relaxed and need not be perfectly balanced. Where these relaxed invariants reduce the Conc-tree list concatenation running time from $O(n)$ to $O(\log n)$, they induce a constant penalty in the popHead operation. Nothing is for free in computer science – the art is in knowing when to pay for what.

In the following, we prove that evaluating a single Spine node per pushHead is sufficient to achieve a constant time execution. We do so by showing that the pushHead operation cannot create two subsequent \$4 suspensions.

Theorem A.15 (Carry Tranquility) *Call a conqueue wing with no two consecutive* \$4 *suspensions a* valid wing*. Assume that we have a sequence of increment steps in a valid wing, in which we pay up to* 1 *carry operations after each increment step. Then, after every carry operation A at any rank $n+2$, there is an increment step B after which there are no suspensions at rank lower than the rank $n+2$, and there is a carry left to spend. Between the steps A and B, there are no suspensions at the depth $n+1$.*

Proof. We inductively prove this by relating the wing to a 4-digit base-2 number.

Observe any digit $y \in \{3, \$4\}$ at rank $n+2$ after a carry operation A:

$$p_n \;\; 2_{n+1} \;\; y_{n+2}$$

We know that the digit at the rank $n+1$ must be 2, or else a carry would not have occurred at $n+2$. Before the next suspension appears at rank $n+1$, a sequence of increment steps must flip the digit at $n+1$ to 3:

$$p'_n \;\; 3_{n+1} \;\; y_{n+2}$$

Since the carry step just occurred at rank $n+1$, the inductive hypothesis now holds for rank $n+1$ and n. If there is a subsequent increment step B after which there are not suspensions at rank lower than the rank $n+1$, and a carry left to spend, then the same holds for the rank $n+2$. □

Next, we show that the popHead operation cannot create an arbitrary long sequence of Zero nodes or unbalanced One nodes.

Theorem A.16 (Borrow Tranquility) *Assume that we have a sequence of decrement steps in a wing, in which we pay up to* 2 *borrow operations after each decrement step.*

Then, after every borrow operation A at any rank n, there is a decrement step B after which there are no `Zero` *trees lower than the rank n, and there is a borrow left to spend. Between the steps A and B, there are no* `Zero` *trees at the depths $n-1$ and $n-2$.*

Proof. The proof is similar to that in Theorem A.15. □

Although a program using a conqueue can invoke any sequence of arbitrary deque operations, note that a `pushHead` is beneficial for borrow suspensions, and `popHead` is beneficial for carry suspensions – either operation just buys additional time to evaluate the opposite kind of a suspension. Orthogonally, to account for operations `pushLast` and `popLast`, we pay the cost of additional 2 suspensions on the opposite wing of the conqueue, as was shown in the pseudocode in Figure 3.10. The following corollary is a direct consequence of Theorem A.15 and Theorem A.16.

Corollary A.17 *Persistent lazy conqueue deque operations run in worst-case $O(1)$ time.*

Finally, the conqueue normalization must run in logarithmic time. By implementing a recursive `wrap` method that concatenates Conc-tree lists from both conqueue wings in an order with strictly increasing heights [Prokopec(2014a)], we achieve an $O(\log n)$ worst-case running time normalization operation by Theorem A.8.

Corollary A.18 *Persistent lazy conqueue concatenation operation runs in worst-case $O(\log \max(n_1, n_2))$ time.*

B Ctrie Proofs

In this appendix we prove several important properties of Ctrie operations. First of all, we prove that any basic Ctrie operation is *safe*. Before we start, we must first define what being safe means.

A data structure in this context is defined as a collection of objects in memory each of which holds some number of pointers to other objects. The allowed configurations of objects and pointers are defined by the invariants of the data structure. We say that a data structure is in a valid state if it respects a specific set of invariants.

Every data structure state $\mathbb{S}$ is consistent with a specific abstract state $\mathbb{A}$. In the context of sets and maps, an abstract state is represented by an abstract set of elements. An abstract data structure operation is defined by changing the abstract state from $\mathbb{A}$ to $\mathbb{A}'$. Every data structure operation implementation must correspond to an abstract operation. A data structure operation implementation (or simply, data structure operation) is correct if it changes the state $\mathbb{S}$ of the data structure to some other state $\mathbb{S}'$ such that $\mathbb{S}$ corresponds to the abstract state $\mathbb{A}$ and $\mathbb{S}'$ corresponds to the abstract state $\mathbb{A}'$.

We will start by defining the Ctrie invariants and the mapping from Ctrie memory layout to the abstract state (set of elements) it represents. We will then do the same for its operations and prove that every Ctrie operation is correct in the way described above.

Linearizability [Herlihy and Wing(1990)] is an important property to prove about the operations on concurrent data structures. An operation executed by some thread is linearizable if the rest of the system observes the corresponding data structure state change occurring instantaneously at a single point in time after the operation started and before it finished.

Lock-freedom [Herlihy and Shavit(2008)] is another important property. An operation executed by some thread is lock-free if and only if at any point of the execution of that operation some (potentially different) thread completes some (potentially different) concurrent operation after a finite number of execution steps. This property guarantees system-wide progress, as at least one thread must always complete its operations. We prove this by showing that every data structure state change is a finite number of execution steps apart from another. This is not sufficient – not every data structure state change necessarily needs to result in abstract state changes, as the mapping between data structure state and the abstract state does not need to be a bijection. Therefore, we will show that in any sequence of operations there can only be finitely many data structure changes that do not result in an abstract state change.

Finally, we will show that the Ctrie data structure is compact – it eventually occupies the amount of memory bound by $O(|\mathbb{A}|)$, where $\mathbb{A}$ is the abstract set it corresponds to.

In what follows, we will simplify the proof by assuming the Ctrie depth is unbound and we never use L-nodes. The existence of L-nodes, however, does not change the structure of this proof, and it can easily be generalized to include L-nodes as well.

The proofs are similar to the previously published proofs for a Ctrie variant with somewhat more complicated invariants [Prokopec et al.(2011a)Prokopec, Bagwell, and Odersky] compared to later work [Prokopec et al.(2012a)Prokopec, Bronson, Bagwell, and Odersky].

Basics. Value W is called the **branching width**. An **I-node** *in* is a node holding a reference *main* to other nodes. A **C-node** *cn* is a node holding a bitmap *bmp* and an set of references to other nodes called *array*. A C-node is **k-way** if $length(cn.array) = k$. An **S-node** *sn* is a node holding a key k and a value v. An S-node can be **tombed**, denoted by $sn\dagger$, meaning its tomb flag is set [1]. A reference $cn.arr(r)$ in the *array* defined as $array(\#(((1 << r) - 1) \odot cn.bmp))$, where $\#$ is the bitcount and $\odot$ is the bitwise-and operation. Any node $n_{l,p}$ is at **level** l if there are l/W C-nodes on the simple path between itself and the root I-node. **Hashcode chunk** of a key k at level l is defined as $m(l,k) = (hashcode(k) >> l) \mod 2^W$. A node at level 0 has a **hashcode prefix** $p = \epsilon$, where ϵ is an empty string. A node n at level $l + W$ has a hashcode prefix $p = q \cdot r$ if and only if it can be reached from the closest parent C-node $cn_{l,q}$ by following the reference $cn_{l,q}.arr(r)$. A reference $cn_{l,p}.sub(k)$ is defined as:

$$cn_{l,p}.sub(k) = \begin{cases} cn_{l,p}.arr(m(l,k)) & \text{if } cn_{l,p}.flg(m(l,k)) \\ null & \text{otherwise} \end{cases}$$

$$cn_{l,p}.flg(r) \Leftrightarrow cn_{l,p}.bmp \odot (1 \ll r) \neq 0$$

Ctrie. A **Ctrie** is defined as the reference *root* to a root node of the trie. A Ctrie **state** $\mathbb{S}$ is defined as the configuration of nodes reachable from the *root* by following references in the nodes. A key is within the configuration if and only if it is in a node reachable

[1] To simplify the proof, we will use this representation for T-nodes from section 4.2.2.

from the root. More formally, the relation $hasKey(in_{l,p}, k)$ on an I-node in at the level l with a prefix p and a key k holds if and only if (several relations for readability):

$$
\begin{aligned}
&holds(in_{l,p}, k) \Leftrightarrow in_{l,p}.main = sn : SNode \wedge sn.k = k \\
&holds(cn_{l,p}, k) \Leftrightarrow cn_{l,p}.sub(k) = sn : SNode \wedge sn.k = k \\
&hasKey(cn_{l,p}, k) \Leftrightarrow holds(cn_{l,p}, k) \vee \\
&(cn_{l,p}.sub(k) = in_{l+w,p \cdot m(l,k)} \wedge hasKey(in_{l+w,p \cdot m(l,k)}, k)) \\
&hasKey(in_{l,p}, k) \Leftrightarrow holds(in_{l,p}, k) \vee \\
&(in_{l,p}.main = cn_{l,p} : CNode \wedge hasKey(cn_{l,p}, k))
\end{aligned}
$$

Definition. We define the following invariants for the Ctrie.

INV1 $inode_{l,p}.main = cnode_{l,p} | snode\dagger$

INV2 $\#(cn.bmp) = length(cn.array)$

INV3 $cn_{l,p}.flg(r) \neq 0 \Leftrightarrow cn_{l,p}.arr(r) \in \{sn, in_{l+W,p \cdot r}\}$

INV4 $cn_{l,p}.arr(r) = sn \Leftrightarrow hashcode(sn.k) = p \cdot r \cdot s$

INV5 $in_{l,p}.main = sn\dagger \Leftrightarrow hashcode(sn.k) = p \cdot r$

Validity. A state $\mathbb{S}$ is **valid** if and only if the invariants INV1-5 are true in the state $\mathbb{S}$.

Abstract set. An **abstract set** $\mathbb{A}$ is a mapping $K \Rightarrow \{\bot, \top\}$ which is true only for those keys which are a part of the abstract set, where K is the set of all keys. An **empty abstract set** $\varnothing$ is a mapping such that $\forall k, \varnothing(k) = \bot$. Abstract set operations are $insert(k, \mathbb{A}) = \mathbb{A}_1 : \forall k' \in \mathbb{A}_1, k' = k \vee k' \in \mathbb{A}$, $lookup(k, \mathbb{A}) = \top \Leftrightarrow k \in \mathbb{A}$ and $remove(k, \mathbb{A}) = \mathbb{A}_1 : k \notin \mathbb{A}_1 \wedge \forall k' \in \mathbb{A}, k \neq k' \Rightarrow k' \in \mathbb{A}$. Operations *insert* and *remove* are **destructive**.

Consistency. A Ctrie state $\mathbb{S}$ is **consistent** with an abstract set $\mathbb{A}$ if and only if $k \in \mathbb{A} \Leftrightarrow hasKey(Ctrie, k)$. A destructive Ctrie operation *op* is **consistent** with the corresponding abstract set operation *op′* if and only if applying *op* to a state $\mathbb{S}$ consistent with $\mathbb{A}$ changes the state into $\mathbb{S}'$ consistent with an abstract set $\mathbb{A}' = op(k, \mathbb{A})$. A Ctrie *lookup* is **consistent** with the abstract set lookup if and only if it returns the same value as the abstract set *lookup*, given that the state $\mathbb{S}$ is consistent with $\mathbb{A}$. A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ of the Ctrie such that $\mathbb{S}$ is consistent with an abstract set $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$ and $\mathbb{A} \neq \mathbb{A}'$.

We point out that there are multiple valid states corresponding to the same abstract set.

Theorem B.1 (Safety) *At all times t, a Ctrie is in a valid state $\mathbb{S}$, consistent with some abstract set $\mathbb{A}$. All Ctrie operations are consistent with the semantics of the abstract set $\mathbb{A}$.*

First, it is trivial to see that if the state $\mathbb{S}$ is valid, then the Ctrie is also consistent with some abstract set $\mathbb{A}$. Second, we prove the theorem using structural induction. As induction base, we take the empty Ctrie which is valid and consistent by definition. The induction hypothesis is that the Ctrie is valid and consistent at some time t. We use the hypothesis implicitly from this point on. Before proving the induction step, we introduce additional definitions and lemmas.

Definition. A node is **live** if and only if it is a C-node, a non-tombed S-node or an I-node whose *main* reference points to a C-node. A **nonlive** node is a node which is not live. A **tomb-I-node** is an I-node with a *main* set to a tombed S-node $sn\dagger$. A node is a **singleton** if it is an S-node or an I-node *in* such that $in.main = sn\dagger$, where $sn\dagger$ is tombed.

Lemma B.2 (End of life) *If an I-node in is a tomb-I-node at some time t_0, then $\forall t > t_0$ in.main is not written.*

Proof. For any I-node *in* which becomes reachable in the Ctrie at some time t, all assignments to *in.main* at any time $t_0 > t$ occur in a CAS instruction – we only have to inspect these writes.

Every CAS instruction on *in.main* is preceded by a check that the expected value of *in.main* is a C-node. From the properties of CAS, it follows that if the current value is a tombed S-node, the CAS will not succeed. Therefore, tomb-I-nodes can be written to *in.main*. □

Lemma B.3 *C-nodes and S-nodes are immutable – once created, they no longer change the value of their fields.*

Proof. Trivial inspection of the pseudocode reveals that k, v, *tomb*, *bmp* and *array* are never assigned a value after an S-node or a C-node was created. □

Definition. A **contraction** *ccn* of a C-node *cn* seen at some time t_0 is a node such that:

- $ccn = sn\dagger$ if $length(cn.array) = 1$ and $cn.array(0)$ is an S-node at t_0

- otherwise, *ccn* is a *cn*

A **compression** *ccn* of a C-node *cn* seen at some time t_0 is the contraction of the C-node obtained from *cn* so that at least those tomb-I-nodes *in* existing at t_0 are resurrected - that is, replaced by untombed copies *sn* of $sn\dagger = in.main$.

Lemma B.4 *Methods toCompressed and toContracted return the compression and the contraction of a C-node cn, respectively.*

Proof. From lemma B.3 we know that a C-node does not change values of *bmp* or *array* once created. From lemma B.2 we know that all the nodes that are nonlive at t_0 must be nonlive $\forall t > t_0$. Methods *toCompressed* or *toContracted* scan the array of *cn* sequentially and make checks which are guaranteed to stay true if they pass – when these methods complete at some time $t > t_0$ they will have resurrected at least those I-nodes that were nonlive at some point t_0 after the operation began. □

Lemma B.5 *Invariants INV1, INV2 and INV3 are always preserved.*

Proof. INV1: I-node creation and every CAS instruction abide this invariant. There are no other writes to *main*.

INV2, INV3: Trivial inspection of the pseudocode shows that the creation of C-nodes abides these invariants. From lemma B.3 we know that C-nodes are immutable. These invariants are ensured during construction and do not change subsequently. □

Lemma B.6 *If any CAS instruction makes an I-node in unreachable from its parent at some time t, then in is nonlive at time t.*

Proof. We will show that all the I-nodes a CAS instruction could have made unreachable from their parents at some point t_1 were nonlive at some time $t_0 < t_1$. The proof then follows directly from lemma B.2. We now analyze successful CAS instructions.

In lines 110 and 122, if *r* is an I-node and it is removed from the trie, then it must have been previously checked to be a a nonlive I-node by a call to *resurrect* in lines 97 and 121, respectively, as implied by lemma B.4.

In lines 33, 48 and 44, a C-node *cn* is replaced with a new C-node *ncn* which contains all the references to I-nodes as *cn* does, and possibly some more. These instructions do not make any I-nodes unreachable.

In line 79, a C-node *cn* is replaced with a new *ncn* which contains all the node references as *cn* but without one reference to an S-node – all the I-nodes remain reachable. □

Corollary B.7 *Lemma B.6 has a consequence that any I-node in can only be made unreachable in the Ctrie through modifications in their immediate parent I-node (or the root reference if in is referred by it). If there is a parent that refers to in, then that parent is live by definition. If the parent had been previously removed, lemma B.6 tells us that the parent would have been nonlive at the time. From lemma B.2 we know that the parent would remain nonlive afterwards. This is a contradiction.*

Lemma B.8 *If at some time t_1 an I-node in is read by some thread (lines 2, 11, 24, 36, 60, 66 followed by a read of C-node $cn = in.main$ in the same thread at time $t_2 > t_1$ (lines 7, 28, 66, 83, 113), then in is reachable from the root at time t_2. Trivially, so is $in.main$.*

Proof. Assume, that I-node *in* is not reachable from the root at t_2. That would mean that *in* was made unreachable at an earlier time $t_0 < t_2$. Corollary B.7 says that *in* was then nonlive at t_0. However, from lemma B.2 it follows that *in* must be nonlive for all times greater than t_0, including t_2. This is a contradiction – *in* is live at t_2, since it contains a C-node $cn = in.main$. □

Lemma B.9 (Presence) *Reading a C-node cn at some time t_0 and then $sn = cn.sub(k)$ such that $k = sn.k$ at some time $t_1 > t_0$ means that the relation $hasKey(root, k)$ holds at time t_0. Trivially, k is then in the corresponding abstract set $\mathbb{A}$.*

Proof. We know from lemma B.8 that the corresponding C-node *cn* was reachable at some time t_0. Lemma B.3 tells us that *cn* and *sn* were the same $\forall t > t_0$. Therefore, *sn* was present in the array of *cn* at t_0, so it was reachable. Furthermore, *sn.k* is the same $\forall t > t_0$. It follows that $hasKey(root, x)$ holds at time t_0. □

Definition. A **longest path** of nodes $\pi(h)$ for some hashcode h is the sequence of nodes from the root to a leaf of a valid Ctrie such that:

- if $root = I_{node}\{main = C_{node}\{array = \varnothing\}\}$ then $\pi(h) = \epsilon$
- otherwise, the first node in $\pi(h)$ is *root*, which is an I-node
- $\forall in \in \pi(h)$ if $in.main = cn \in C_{node}$, then the next element in the path is *cn*
- $\forall sn \in \pi(h)$ if $sn \in S_{node}$, then the last element in the path is *sn*
- $\forall cn_{l,p} \in \pi(h), h = p \cdot r \cdot s$ if $cn.flg(r) = \bot$, then the last element in the path is *cn*, otherwise the next element in the path is $cn.arr(r)$

Lemma B.10 (Longest path) *Assume that a non-empty Ctrie is in a valid state at some time t. The longest path of nodes $\pi(h)$ for some hashcode $h = r_0 \cdot r_1 \cdots r_n$ is a sequence $in_{0,\epsilon} \rightarrow cn_{0,\epsilon} \rightarrow in_{W,r_0} \rightarrow \ldots \rightarrow in_{W \cdot m, r_0 \cdots r_m} \rightarrow x$, where, with $\forall z, cn_z \in C_{node}$ and $\forall, sn_z \in S_{node}$, $x \in \{cn_{W \cdot m, r_0 \cdots r_m}, sn, cn_{W \cdot m, r_0 \cdots r_m} \rightarrow sn\}$.*

Proof. Trivially from the invariants and the definition of the longest path. □

Lemma B.11 (Absence I) *Assume that at some time t_0 $\exists cn = in.main$ for some node $in_{l,p}$ and the algorithm is searching for a key k. Reading a C-node cn at some time t_0 such that $cn.sub(k) = null$ and $hashcode(k) = p \cdot r \cdot s$ implies that the relation $hasKey(root, k)$ does not hold at time t_0. Trivially, k is not in the corresponding abstract set $\mathbb{A}$.*

Proof. Lemma B.8 implies that in is in the configuration at time t_0, because $cn = cn_{l,p}$ such that $hashcode(k) = p \cdot r \cdot s$ is live. The induction hypothesis states that the Ctrie was valid at t_0. We prove that $hasKey(root, k)$ does not hold by contradiction. Assume there exists an S-node sn such that $sn.k = k$. By lemma B.10, sn can only be the last node of the longest path $\pi(h)$, and we know that cn is the last node in $\pi(h)$. □

Lemma B.12 (Absence II) *Assume that the algorithm is searching for a key k. Reading a live S-node sn at some time t_0 and then $x = sn.k \neq k$ at some time $t_1 > t_0$ means that the relation $hasKey(root, x)$ does not hold at time t_0. Trivially, k is not in the corresponding abstract set $\mathbb{A}$.*

Proof. Contradiction similar to the one in the previous lemma. □

Lemma B.13 (Fastening) *1. Assume that one of the CAS instructions in lines 33 and 44 succeeds at time t_1 after $in.main$ was read in line 28 at time t_0. The $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ does not hold.*

2. Assume that the CAS instruction in line 48 succeeds at time t_1 after $in.main$ was read in line 28 at time t_0. The $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ holds.

3. Assume that the CAS instruction in line 79 succeeds at time t_1 after $in.main$ was read in line 66 at time t_0. The $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ holds.

Proof. The algorithm never creates a reference to a newly allocated memory areas unless that memory area has been previously reclaimed. Although it is possible to extend the pseudocode with memory management directives, we omit memory-reclamation from the pseudocode and assume the presence of a garbage collector which does not reclaim memory areas as long as there are references to them reachable from the program. In the pseudocode, CAS instructions always work on memory locations holding references – $CAS(x, r, r')$ takes a reference r to a memory area allocated for nodes as its expected value, meaning that a reference r that is reachable in the program exists from the time t_0 when it was read until $CAS(x, r, r')$ was invoked at t_1. On the other hand, the new value for the CAS is in all cases a newly allocated object. In the presence of a garbage collector with the specified properties, a new object cannot be allocated in any of the areas still being referred to. It follows that if a CAS succeeds at time t_1, then $\forall t, t_0 \leq t < t_1$, where t_0 is the time of reading a reference and t_1 is the time when CAS occurs, the corresponding memory location x had the same value r.

We now analyze specific cases from the lemma statement:

1. From lemma B.10 we know that for some hashcode $h = hashcode(k)$ there exists a longest path of nodes $\pi(h) = in_{0,\epsilon} \rightarrow \ldots \rightarrow cn_{l,p}$ such that $h = p \cdot r \cdot s$ and that sn such that $sn.k = k$ cannot be a part of this path – it could only be referenced by $cn_{l,p}.sub(k)$ of the last C-node in the path. We know that $\forall t, t_0 \leq t < t_1$ reference cn points to the same C-node. We know from B.3 that C-nodes are immutable. In the case of line 33, the check to $cn.bmp$ preceding the CAS ensures that $\forall t, t_0 \leq t < t_1$ $cn.sub(k) = null$. In the case of line 44, there is a preceding check that the key k is not contained in sn. We know from B.6 that cn is reachable during this time, because in is reachable. Therefore, $hasKey(root, k)$ does not hold $\forall t, t_0 \leq t < t_1$.

2., 3. We know that $\forall t, t_0 \leq t < t_1$ reference cn points to the same C-node. C-node cn is reachable as long as its parent I-node in is reachable. We know that in is reachable by lemma B.6, since in is live $\forall t, t_0 \leq t < t_1$. We know that cn is immutable by lemma B.3 and that it contains a reference to sn such that $sn.k = k$. Therefore, sn is reachable and $hasKey(root, k)$ holds $\forall t, t_0 \leq t < t_1$. □

Lemma B.14 *Assume that the Ctrie is valid and consistent with some abstract set* $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$. *CAS instructions from lemma B.13 induce a change into a valid state which is consistent with the abstract set semantics.*

Proof. Observe a successful CAS in line 33 at some time t_1 after cn was read in line 28 at time $t_0 < t_1$. From lemma B.13 we know that $\forall t, t_0 \leq t < t_1$, relation $hasKey(root, k)$ does not hold. If the last CAS instruction occuring anywhere in the Ctrie before the CAS in line 33 was at $t_\delta = t_1 - \delta$, then we know that $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the $hasKey$ relation on all keys does not change. We know that at t_1 cn is replaced with a new C-node with a reference to a new S-node sn such that $sn.k = k$, so at t_1 relation $hasKey(root, k)$ holds. Consequently, $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the Ctrie is consistent with an abstract set $\mathbb{A}$ and at t_1 it is consistent with an abstract set $\mathbb{A} \cup \{k\}$. Validity is trivial.

Proofs for the CAS instructions in lines 44, 48 and 79 are similar. □

Lemma B.15 *Assume that the Ctrie is valid and consistent with some abstract set* $\mathbb{A}$ $\forall t, t_1 - \delta < t < t_1$*. If one of the operations clean or cleanParent succeeds with a CAS at* t_1*, the Ctrie will remain valid and consistent with the abstract set* $\mathbb{A}$ *at* t_1*.*

Proof. Operations *clean* and *cleanParent* are atomic - their linearization point is the first successful CAS instruction occuring at t_1. We know from lemma B.4 that methods *toCompressed* and *toContracted* produce a compression and a contraction of a C-node, respectively.

We first prove the property $\exists k, hasKey(n, k) \Rightarrow hasKey(f(n), k)$, where f is either a compression or a contraction. Omitting a tomb-I-node may omit exactly one key, but $f = toCompressed$ resurrects copies sn of removed I-nodes in such that $in.main = sn\dagger$. The $f = toContracted$ replaces a cn with a single key with an sn with the same key. Therefore, the *hasKey* relation is exactly the same for both n and $f(n)$.

We only have to look at cases where CAS instructions succeed. If CAS in line 110 at time t_1 succeeds, then $\forall t, t_0 < t < t_1$ $in.main = cn$ and at t_1 $in.main = toCompressed(cn)$. Assume there is some time $t_\delta = t_1 - \delta$ at which the last CAS instruction in the Ctrie occuring before the CAS in line 110 occurs. Then $\forall t, \max(t_0, t_\delta) \leq t < t_1$ the *hasKey* relation does not change. Additionally, it does not change at t_1, as shown above. Therefore, the Ctrie remains consistent with the abstract set $\mathbb{A}$. Validity is trivial.

Proof for *cleanParent* is similar. □

Corollary B.16 *From lemmas B.14 and B.15 it follows that invariants INV4 and INV5 are always preserved.*

Safety. We proved at this point that the algorithm is safe - Ctrie is always in a valid (lemma B.5 and corollary B.16) state consistent with some abstract set. All operations are consistent with the abstract set semantics (lemmas B.9, B.11, B.12 B.14 and B.15). □

Theorem B.17 (Linearizability) *Operations insert, lookup and remove are linearizable.*

Linearizability. An operation is linearizable if we can identify its linearization point. The linearization point is a single point in time when the consistency of the Ctrie changes. The CAS instruction itself is linearizable, as well as atomic reads. It is known that a single invocation of a linearizable instruction has a linearization point.

1. We know from lemma B.15 that operation *clean* does not change the state of the corresponding abstract set. Operation *clean* is followed by a restart of the operation it was called from and is not preceded by a consistency change – all successful writes in the *insert* and *iinsert* that change the consistency of the Ctrie result in termination.

CAS in line 33 that succeeds at t_1 immediately returns. By lemma B.14, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an empty abstract set $\mathbb{A}$ and at t_1 it is consistent with $\mathbb{A} \cup \{k\}$. If this is the first invocation of *iinsert*, then the CAS is the first and the last write with consistent semantics. If *iinsert* has been recursively called, then it was preceded by an *insert* or *iinsert*. We have shown that if its preceded by a call to *insert*, then there have been no preceding consistency changes. If it was preceded by *iinsert*, then there has been no write in the previous *iinsert* invocation. Therefore, it is the linearization point.

Similar arguments hold for CAS instructions in lines 48 and 44. It follows that if some CAS instruction in the *insert* invocation is successful, then it is the only successful CAS instruction. Therefore, *insert* is linearizable.

2. Operation *clean* is not preceded by a write that results in a consistency change and does not change the consistency of the Ctrie.

Assume that a node m is read in line 7 at t_0. By the immutability lemma B.3, if $cn.sub(k) = null$ at t_0 then $\forall t, cn.sub(k) = null$. By corollary B.7, cn is reachable at t_0, so at t_0 the relation $hasKey(root, k)$ does not hold. The read at t_0 is not preceded by a consistency changing write and followed by a termination of the *lookup* so it is a linearization point if the method returns in line 10. By similar reasoning, if the operation returns after line 14, the read in line 7 is the linearization point.

We have identified linearization points for the *lookup*, therefore *lookup* is linearizable.

3. Again, operation *clean* is not preceded by a write that results in a consistency change and does not change the consistency of the Ctrie.

By lemma B.15 operations *clean* and *cleanParent* do not cause a consistency change.

Assume that a node m is read in line 66 at t_0. By similar reasoning as with *lookup* above, the read in line 66 is a linearization point if the method returns in either of the lines 69 or 75.

Assume that the CAS in line 79 succeeds at time t_0. By lemma B.14, $\exists \delta > 0 \forall t, t_1 - \delta < t < t_1$ the Ctrie is consistent with an abstract set $\mathbb{A}$ and at t_1 it is consistent with $\mathbb{A} \setminus \{k\}$. This write is not preceded by consistency changing writes and followed only by *cleanParent* which also do not change consistency. Therefore, it is a linearization point.

We have identified linearization points for the *remove*, so *remove* is linearizable. □

Definition. Assume that a multiple number of threads are invoking a concurrent operation *op*. The concurrent operation *op* is **lock-free** if and only if after a finite number of thread execution steps some thread completes the operation.

Theorem B.18 (Lock-freedom) *Ctrie operations insert, lookup and remove are lock-free.*

The rough idea of the proof is the following. To prove lock-freedom we will first show that there is a finite number of steps between state changes. Then we define a reduction of the space of possible states and show that there can only be finitely many successful CAS instructions which do not result in a consistency change. We have shown in lemmas B.14 and B.15 that only CAS instructions in lines 110 and 122 do not cause a consistency change. We proceed by introducing additional definitions and proving the necessary lemmas. In all cases, we assume there has been no state change which is a consistency change, otherwise that would mean that some operation was completed.

Lemma B.19 *The root is never a tomb-I-node.*

Proof. A tomb S-node can only be assigned to *in.main* of some I-node *in* in *clean*, *cleanParent* and *iremove* in line 79. Neither *clean* nor *cleanParent* are ever called for the *in* in the root of the Ctrie, as they are preceded by the check if *parent* is different than *null*. The CAS in line 79 replaces the current C-node with the result of *toContracted*, which in turn produces a tomb S-node only if the level is greater than 0. □

Lemma B.20 *If a CAS in one of the lines 33, 48, 44, 79, 110 or 122 fails at some time t_1, then there has been a state change since the time t_0 when a respective read in one of the lines 28, 66, 109 or 113 occured. Trivially, the state change preceded the CAS by a finite number of execution steps.*

Proof. The configuration of nodes reachable from the root has changed, since the corresponding *in.main* has changed. Therefore, the state has changed by definition. □

Lemma B.21 *In each operation there is a finite number of execution steps between 2 consecutive calls to a CAS instruction.*

Proof. The *ilookup* and *iinsert* operations have a finite number of executions steps. There are no loops in the pseudocode for *ilookup* in *iinsert*, the recursive calls to them occur on the lower level of the trie and the trie depth is bound – no non-consistency changing CAS increases the depth of the trie.

The *lookup* operation is restarted if and only if *clean* (which contains a CAS) is called in *ilookup*. Due to this and the trie depth bound, there are a finite number of execution steps between two calls to a CAS instruction.

The *insert* operation is restarted if and only if *clean* (which contains a CAS) is called in *iinsert* or due to a preceding failed CAS in lines 33, 48 or 44.

In *iremove*, CAS instructions in calls to *clean* are a finite number of execution steps apart, as before. Method *cleanParent* contain no loops, but is recursive. In case it restarts itself, a CAS is invoked at least once. Between these CAS instructions there is a finite number of execution steps.

A similar analysis as for *lookup* above can be applied to the first phase of *remove* which consists of all the execution steps preceding a successful CAS in line 79.

All operations have a finite number of executions steps between consecutive CAS instructions, assuming that the state has not changed since the last CAS instruction. □

Corollary B.22 *The consequence of lemmas B.21 and B.20 is that there is a finite number of execution steps between two state changes. At any point during the execution of the operation we know that the next CAS instruction is due in a finite number of execution steps (lemma B.21). From lemmas B.14 and B.15 we know that if a CAS succeeds, it changes the state. From lemma B.20 we know that if the CAS fails, the state was changed by someone else.*

We remark at this point that corollary B.22 does not imply that there is a finite number of execution steps between two operations. A state change is not necessarily a consistency change.

Definition. Let there at some time t_0 be a 1-way C-node cn such that $cn.array(0) = in$ and $in.main = sn\dagger$ where $sn\dagger$ is tombed or, alternatively, cn is a 0-way node. We call such cn a **single tip of length** 1. Let there at some time t_0 be a 1-way C-node cn such that $cn.array(0) = I_{node}\{main = cn'\}$ and cn' is a single tip of length k. We call such cn a **single tip of length** $k + 1$.

Definition. The **total path length** d is the sum of the lengths of all the paths from the root to some leaf.

Definition. Assume the Ctrie is in a valid state. Let t be the number of reachable tomb-I-nodes in this state, l the number of live I-nodes, r the number of single tips of any length and d the total path length. We denote the state of the Ctrie as $\mathbb{S}_{t,l,r,d}$. We call the state $\mathbb{S}_{0,l,r,d}$ the **clean** state.

Lemma B.23 *Observe all CAS instructions which never cause a consistency change and assume they are successful. Assuming there was no state change since reading in prior to calling clean, the CAS in line 110 changes the state of the Ctrie from the state* $\mathbb{S}_{t,l,r,d}$ *to* $\mathbb{S}_{t-j,l,r,d' \leq d}$ *where* $j > 0$ *and* $t \geq j$.

Furthermore, the CAS in line 122 changes the state from $\mathbb{S}_{t,l,r,d}$ *to* $\mathbb{S}_{t-1,l,r-j,d-1}$ *where* $t \geq 1$, $d \geq 1$ *and* $j \in \{0, 1\}$.

Proof. We have shown in lemma B.15 that the CAS in line 110 does not change the number of live nodes. In lemma B.4 we have shown that *toCompressed* returns a compression of the C-node *cn* which replaces *cn* at *in.main* at time t.

Provided there is at least one single tip immediately before time t, the compression of the C-node *cn* can omit at most one single tip, decreasing r by one. Omitting a single tip will also decrease d by one.

This proves the statement for CAS in line 110. The CAS in line 122 can be proved by applying a similar line of reasoning. □

Lemma B.24 *If the Ctrie is in a clean state and n threads are executing operations on it, then some thread will execute a successful CAS resulting in a consistency change after a finite number of execution steps.*

Proof. Assume that there are $m \leq n$ threads in the *clean* operation or in the cleanup phase of the *remove*. The state is clean, so there are no nonlive I-nodes – none of the m threads performing *clean* will invoke a CAS after their respective next (unsuccessful) CAS. This means that these m threads will either complete in a finite number of steps or restart the original operation after a finite number of steps. From this point on, as shown in lemma B.21, the first CAS will be executed after a finite number of steps. Since the state is clean, there are no more nonlive I-nodes, so *clean* will not be invoked. Therefore, the first subsequent CAS will result in a consistency change. Since it is the first CAS, it will also be successful. □

Lock-freedom. Assume we start in some state $\mathbb{S}_{t,l,r,d}$. We prove there are a finite number of state changes before reaching a clean state by contradiction. Assume there is an infinite sequence of state changes. We now use results from lemma B.23. In this infinite sequence, a state change which decreases d may occur only finitely many times, since no state change increases d. After this finitely many state changes $d = 0$ so the sequence can contain no more state changes which decrease d. We apply the same reasoning to r – no available state change can increase the value of r, so after finitely many steps $r = 0$. Therefore, the assumption is wrong – such an infinite sequence of state changes does not exist.

From corollary B.22 there are a finite number of execution steps between state changes, so there are a finite number of execution steps before reaching a clean state. By lemma B.24, if the Ctrie is in a clean state, then there are an additional finite number of steps until a consistency change occurs.

This proves that some operation completes after a finite number of steps, so all Ctrie operations are lock-free. □

Theorem B.25 (Compactness) *Assume all remove operations have completed execution. Then there is at most* 1 *single tip of length* 1 *in the trie.*

Compactness. We will prove that if there is a non-root single tip of length 1 in the trie on the longest path $\pi(h)$ of some hashcode h, then there is at least one *remove* operation in the cleanup phase working on the longest path $\pi(h)$.

By contradiction, assume that there are no *remove* operations on the longest path $\pi(h)$ ending with a single tip of length 1. A single tip *cn* can be created at time t_0 by a *remove* invoking a CAS instruction in line 79, by a *clean* invoking a CAS instruction in line 110 or by a *cleanParent* invoking a CAS instruction in line 122. A *clean* operation can only create a single tip of length 1 in line 110 by removing another single tip of length 1, which would imply that there was no *remove* operation on the longest path $\pi(h)$ in the cleanup phase before that *clean*. Similarly *cleanParent* can only create a single tip of length 1 in line 122 by removing another single tip of length 1, implying there were no *remove* operation on the longest path $\pi(h)$ before. This is a contradiction, because only the *remove* operation can call *cleanParent*. A *remove* operation can create a single tip of length 1 where without removing another one in line 79, but that puts the *remove* operation in the cleanup phase, which again leads to a contradiction.

Now, we have to show that if the *cleanParent* ends, the corresponding single tip was removed. If upon reading the state of *in* at t_0 in line 113, the *cleanParent* does not attempt or fails to remove the tip with a CAS at $t_1 > t_0$ in line 122, then the tip was already removed at this level of the trie by some *clean* method after the read in line 122. This is because all other CAS instructions fail if the S-node is tombed, and all operations call *clean* if their read in lines 7, 28 and 66 detected a tombed S-node. Alternatively, the *cleanParent* removes the tip with a successful CAS in line 122. In both cases, a CAS at $t_1 > t_0$ removing the single tip can introduce a single tip at the parent of *in* was a single tip of length 2 at t_1. If so, *remove* calls *cleanParent* at the parent of *in* recursively.

The single tip may either be a 0-way C-node, meaning it is the root, or be a C-node with a single tombed S-node. This proves the initial statement. □

C FlowPool Proofs

In this appendix we prove the correctness of the proposed flow pool implementation, and identify the linearization points for its operations, proving that the operations are linearizable, as discussed in Appendix B. For the lock-freedom proof we refer the reader to related work [Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky].

Data types. A **Block** b is an object which contains an array $b.array$, which itself can contain elements, $e \in Elem$, where **Elem** represents the type of e and can be any countable set. A given block b additionally contains an index $b.index$ which represents an index location in $b.array$, a unique index identifying the array $b.blockIndex$, and $b.next$, a reference to a successor block c where $c.blockIndex = b.blockIndex + 1$. A **Terminal** $term$ is a sentinel object, which contains an integer $term.sealed \in \{-1\} \cup \mathbb{N}_0$, and $term.callbacks$, a set of functions $f \in Elem \Rightarrow Unit$.

We define the following functions:

$$
\begin{aligned}
following(b : Block) &= \begin{cases} \emptyset & \text{if b.next = null,} \\ b.next \cup following(b.next) & \text{otherwise} \end{cases} \\
reachable(b : Block) &= \{b\} \cup following(b) \\
last(b : Block) &= b' : b' \in reachable(b) \wedge b'.next = null \\
size(b : Block) &= |\{x : x \in b.array \wedge x \in Elem\}|
\end{aligned}
$$

Based on them we define the following relation:

$$reachable(b, c) \Leftrightarrow c \in reachable(b)$$

FlowPool. A **FlowPool** *pool* is an object that has a reference *pool.start*, to the first block b_0 (with $b_0.blockIndex = 0$), as well as a reference *pool.current*. We sometimes refer to these just as *start* and *current*, respectively.

A **scheduled callback invocation** is a pair (f, e) of a function $f \in Elem => Unit$ and an element $e \in Elem$. The programming construct that adds such a pair to the set of $futures$ is `future { f(e) }`.

The **FlowPool state** is defined as a pair of the directed graph of objects transitively reachable from the reference *start* and the set of scheduled callback invocations called $futures$.

A **state changing** or **destructive** instruction is any atomic write or CAS instruction that changes the FlowPool state.

We say that the FlowPool **has an element** e at some time t_0 if and only if the relation $hasElem(start, e)$ holds.

$$hasElem(start, e) \Leftrightarrow \exists b \in reachable(start), e \in b.array$$

We say that the FlowPool **has a callback** f at some time t_0 if and only if the relation $hasCallback(start, f)$ holds.

$$\begin{aligned} hasCallback(start, f) \quad \Leftrightarrow \quad & \forall b = last(start), b.array = x^P \cdot t \cdot y^N, x \in Elem, \\ & t = Terminal(seal, callbacks), f \in callbacks \end{aligned}$$

We say that a callback f in a FlowPool **will be called** for the element e at some time t_0 if and only if the relation $willBeCalled(start, e, f)$ holds.

$$willBeCalled(start, e, f) \Leftrightarrow \exists t_1, \forall t > t_1, (f, e) \in futures$$

We say that the FlowPool is **sealed** at the size s at some t_0 if and only if the relation $sealedAt(start, s)$ holds.

$$\begin{aligned} sealedAt(start, s) \quad \Leftrightarrow \quad & s \neq -1 \wedge \forall b = last(start), b.array = x^P \cdot t \cdot y^N, \\ & x \in Elem, t = Terminal(s, callbacks) \end{aligned}$$

FlowPool operations are `append`, `foreach` and `seal`, and are defined by pseudocodes in Figures 4.7 and 4.8.

Invariants. We define the following invariants for the **FlowPool**:

INV1 $start = b : Block, b \neq null, current \in reachable(start)$

INV2 $\forall b \in reachable(start), b \notin following(b)$

INV3 $\forall b \in reachable(start), b \neq last(start)$
$\Rightarrow size(b) = LASTELEMPOS \wedge b.array(BLOCKSIZE - 1) \in Terminal$

INV4 $\forall b = last(start), b.array = p \cdot c \cdot n$, where:

$p = X^P, c = c_1 \cdot c_2, n = null^N$

$x \in Elem, c_1 \in Terminal, c_2 \in \{null\} \cup Terminal$

$P + N + 2 = BLOCKSIZE$

INV5 $\forall b \in reachable(start), b.index > 0 \Rightarrow b.array(b.index - 1) \in Elem$

Validity. A FlowPool state $\mathbb{S}$ is **valid** if and only if the invariants [INV1-5] hold for that state.

Abstract pool. An **abstract pool** $\mathbb{P}$ is a function from time t to a tuple $(elems, callbacks, seal)$ such that:

$$seal \in \{-1\} \cup \mathbb{N}_0$$

$$callbacks \subset \{(f : Elem \Rightarrow Unit, called)\}$$

$$called \subseteq elems \subseteq Elem$$

We say that an abstract pool $\mathbb{P}$ **is in state** $\mathbb{A} = (elems, callbacks, seal)$ at time t if and only if $\mathbb{P}(t) = (elems, callbacks, seal)$.

Abstract pool operations. We say that an **abstract pool operation** op that is applied to some abstract pool $\mathbb{P}$ in abstract state $\mathbb{A}_0 = (elems_0, callbacks_0, seal_0)$ at some time t **changes** the abstract state of the abstract pool to $\mathbb{A} = (elems, callbacks, seal)$ if $\exists t_0, \forall \tau, t_0 < \tau < t, \mathbb{P}(\tau) = \mathbb{A}_0$ and $\mathbb{P}(t) = \mathbb{A}$. We denote this as $\mathbb{A} = op(\mathbb{A}_0)$.

Abstract pool operation $foreach(f)$ changes the abstract state at t_0 from $(elems, callbacks, seal)$ to $(elems, (f, \emptyset) \cup callbacks, seal)$. Furthermore:

$$\begin{aligned} \exists t_1 \geq t_0, \quad & \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, callbacks_2, seal_2) \\ & \wedge \forall (f, called_2) \in callbacks_2, elems \subseteq called_2 \subseteq elems_2 \end{aligned}$$

Abstract pool operation $append(e)$ changes the abstract state at t_0 from $(elems, callbacks, seal)$ to $(\{e\} \cup elems, callbacks, seal)$. Furthermore:

$$\begin{aligned} \exists t_1 \geq t_0, \quad & \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, callbacks_2, seal_2) \\ & \wedge \forall (f, called_2) \in callbacks_2, (f, called) \in callbacks \Rightarrow e \in called_2 \end{aligned}$$

Abstract pool operation $seal(s)$ changes the abstract state of the FlowPool at t_0 from $(elems, callbacks, seal)$ to $(elems, callbacks, s)$, assuming that $seal \in \{-1\} \cup \{s\}$ and $s \in \mathbb{N}_0$, and $|elems| \leq s$.

Consistency. A FlowPool state $\mathbb{S}$ is **consistent** with an abstract pool $\mathbb{P} = (elems, callbacks, seal)$ at t_0 if and only if $\mathbb{S}$ is a valid state and:

$$\forall e \in Elem, hasElem(start, e) \Leftrightarrow e \in elems$$

$\forall f \in Elem => Unit, hasCallback(start, f) \Leftrightarrow f \in callbacks$

$\forall f \in Elem => Unit, \forall e \in Elem, willBeCalled(start, e, f) \Leftrightarrow \exists t_1 \geq t_0, \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, (f, called_2) \cup callbacks_2, seal_2), elems \subseteq called_2$

$\forall s \in \mathbb{N}_0, sealedAt(start, s) \Leftrightarrow s = seal$

A FlowPool operation *op* is **consistent** with the corresponding abstract state operation op' if and only if $\mathbb{S}' = op(\mathbb{S})$ is consistent with an abstract state $\mathbb{A}' = op'(\mathbb{A})$.

A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ such that $\mathbb{S}$ is consistent with an abstract state $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$, where $\mathbb{A} \neq \mathbb{A}'$.

Proposition C.1 *Every valid state is consistent with some abstract pool.*

Theorem C.2 (Safety) *FlowPool operation* `create` *creates a new FlowPool consistent with the abstract pool* $\mathbb{P} = (\emptyset, \emptyset, -1)$*. FlowPool operations* `foreach`*,* `append` *and* `seal` *are consistent with the abstract pool semantics.*

Lemma C.3 (End of life) *For all blocks* $b \in reachable(start)$*, if value* $v \in Elem$ *is written to b.array at some position idx at some time* t_0*, then* $\forall t > t_0, b.array(idx) = v$*.*

Proof. The CAS in line 20 is the only CAS which writes an element. No other CAS has a value of type *Elem* as the expected value. This means that once the CAS in line 20 writes a value of type *Elem*, no other write can change it.

Corollary C.4 *The end of life lemma implies that if all the values in b.array are of type Elem at* t_0*, then* $\forall t > t_0$ *there is no write to b.array.*

Lemma C.5 (Valid hint) *For all blocks* $b \in reachable(start)$*, if* $b.index > 0$ *at some time* t_0*, then* $b.array(b.index - 1) \in Elem$ *at time* t_0*.*

Proof. Observe every write to *b.index* – they are all unconditional. However, at every such write occurring at some time t_1 that writes some value *idx* we know that some previous value at *b.array* entry $idx - 1$ at some time $t_0 < t_1$ was of type *Elem*. Hence, from C.3 it follows that $\forall t \geq t_1, b.array(idx - 1) \in Elem$.

Corollary C.6 (Compactness) *For all blocks* $b \in reachable(start)$*, if for some idx* $b.array(idx) \in Elem$ *at time* t_0 *then* $b.array(idx - 1) \in Elem$ *at time* t_0*. This follows directly from the C.3 and C.5, and the fact that the CAS in line 20 only writes to array entries idx for which it previously read the value from b.index.*

Transition. If for a function $f(t)$ there exist times t_0 and t_1 such that $\forall t, t_0 < t < t_1, f(t) = v_0$ and $f(t_1) = v_1$, then we say that the function f goes through a **transition** at t_1. We denote this as:

$f : v_0 \xrightarrow{t_1} v_1$

Or, if we don't care about the exact time t_1, simply as:

$f : v_0 \rightarrow v_1$

Monotonicity. A function of time $f(t)$ is said to be **monotonic**, if every value in its string of transitions occurs only once.

Lemma C.7 (Freshness) *For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x is monotonic.*

Proof. CAS instruction in line 20 writes a value of type *Elem*. No CAS instruction has a value of type *Elem* as the expected value, so this write occurs only once.

Trivial analysis of CAS instructions in lines 93 and 113, shows that their expected values are of type $Terminal$. Their new values are always freshly allocated.

The more difficult part is to show that CAS instruction in line 19 respects the statement of the lemma.

Since the CAS instructions in lines 93 and 113 are preceeded by a read of $idx = b.index$, from C.5 it follows that $b.array(idx - 1)$ contains a value of type *Elem*. These are also the only CAS instructions which replace a $Terminal$ value with another $Terminal$ value. The new value is always unique, as shown above.

So the only potential CAS to write a non-fresh value to $idx + 1$ is the CAS in line 19.

A successful CAS in line 19 overwrites a value cb_0 at $idx + 1$ read in line 16 at t_0 with a new value cb_2 at time t_2. Value cb_2 was read in line 17 at t_1 from the entry idx. The string of transitions of values at idx is composed of unique values at least since t_1 (by C.3), since there is a value of type *Elem* at the index $idx - 1$.

The conclusion above ensures that the values read in line 17 to be subsequently used as new values for the CAS in line 19 form a monotonic function $f(t) = b.array(idx)$ at t.

Now assume that a thread T1 successfully overwrites cb_0 via CAS in line 19 at $idx + 1$ at time t_2 to a value cb_2 read from idx at t_1, and that another thread T2 is the **first** thread (since the FlowPool was created) to subsequently successfully complete the CAS in line 19 at $idx + 1$ at time $t_{prev2} > t_2$ with some value cb_{prev2} which was at $idx + 1$ at some time $t < t_0$.

That would mean that $b.array(idx + 1)$ does not change during $\langle t_0, t_2 \rangle$, since T2 was the first thread the write a non-fresh value to $idx + 1$, and any other write would cause the CAS in line 19 by T1 to fail.

Also, that would mean that the thread T2 read the value cb_{prev2} in line 17 at some time $t_{prev1} < t_1$ and successfully completed the CAS at time $t_{prev2} > t_2$. If the CAS was successful, then the read in line 16 by T2 occured at $t_{prev0} < t_{prev1} < t_1$. Since we assumed that T2 is the first thread to write a value cb_{prev2} to $idx + 1$ at time t_{prev2} which was previously in $idx + 1$ at some time $t < t_0$, then the CAS in line 19 at time t_{prev2} could not have succeeded, since its expected value is cb_{prev0} read at some time t_{prev0}, and we know that the value at $idx + 1$ was changed at least once in $\langle t_{prev0}, t_{prev2} \rangle$ because of the write of a fresh value by thread T1 at $t_2 \in \langle t_{prev0}, t_{prev2} \rangle$. This value is known to be fresh because $b.array(idx)$ is a monotonic function at least since t_{prev1}, and the read of the new value written by T1 occurred at $t_1 > t_{prev1}$. We also know that there is no other thread T3 to write the value cb_{prev0} during $\langle t_{prev0}, t_{prev2} \rangle$ back to $idx + 1$, since we assumed that T2 is the first to write a non-fresh value at that position.

Hence, a contradiction shows that there is no thread T2 which is the **first** to write a non-fresh value via CAS in line 19 at $idx + 1$ for any idx, so there is no thread that writes a non-fresh value at all.

Lemma C.8 (Lifecycle) *For all blocks $b \in reachable(start)$, and for all $x \in b.array$, function x goes through and only through the prefix of the following transitions:*

$null \rightarrow cb_1 \rightarrow \cdots \rightarrow cb_n \rightarrow elem$, where:

$cb_i \in Terminal, i \neq j \Rightarrow cb_i \neq cb_j, elem \in Elem$

Proof. First of all, it is obvious from the code that each block that becomes an element of $reachable(start)$ at some time t_0 has the value of all $x \in b.array$ set to $null$.

Next, we inspect all the CAS instructions that operate on entries of $b.array$.

The CAS in line 20 has a value $curo \in Terminal$ as an expected value and writes an $elem \in Elem$. This means that the only transition that this CAS can cause is of type $cb_i \in Terminal \rightarrow elem \in Elem$.

We will now prove that the CAS in line 19 at time t_2 is successful if and only if the entry at $idx + 1$ is $null$ or $nexto \in Terminal$. We know that the entry at $idx + 1$ does not change $\forall t, t_0 < t < t_2$, where t_0 is the read in line 16, because of C.7 and the fact that CAS in line 19 is assumed to be successful. We know that during the read in line 17 at time t_1, such that $t_0 < t_1 < t_2$, the entry at idx was $curo \in Terminal$, by trivial analysis of the `check` procedure. It follows from corollary C.6 that the array entry $idx + 1$ is not of type $Elem$ at time t_1, otherwise array entry idx would have to be of type $Elem$. Finally, we know that the entry at $idx + 1$ has the same value during the interval $\langle t_1, t_2 \rangle$, so its value is not $Elem$ at t_2.

The above reasoning shows that the CAS in line 19 always overwrites a one value of type $Terminal$ (or $null$) with another value of type $Terminal$. We have shown in C.7 that it never overwrites the value cb_0 with a value cb_2 that was at $b.array(idx)$ at an earlier time.

Finally, note that the statement for CAS instructions in lines 93 and 113 also follows directly from the proof for C.7.

Lemma C.9 (Subsequence) *Assume that for some block $b \in reachable(start)$ the transitions of $b.array(idx)$ are:*

$$b.array(idx) : null \to cb_1 \to \cdots \to cb_n \overset{t_0}{\to} elem : Elem$$

Assume that the transitions of $b.array(idx + 1)$ up to time t_0 are:

$$b.array(idx + 1) : null \to cb'_1 \to \cdots \to cb'_m$$

The string of transitions $null \to cb'_1 \to \cdots \to cb'_m$ is a subsequence of $null \to cb_1 \to \cdots \to cb_n$.

Proof. Note that all the values written to $idx + 1$ before t_0 by CAS in line 19 were previously read from idx in line 17. This means that the set of values occurring in $b.array(idx + 1)$ before t_0 is a subset of the set of values in $b.array(idx)$. We have to prove that it is actually a subsequence.

Assume that there exist two values cb_1 and cb_2 read by threads T1 and T2 in line 17 at times t_1 and $t_2 > t_1$, respectively. Assume that these values are written to $idx + 1$ by threads T1 and T2 in line 19 in the opposite order, that is at times t_{cas1} and $t_{cas2} < t_{cas1}$, respectively. That would mean that the CAS by thread T1 would have to fail, since its expected value cb_0 has changed between the time it was read in line 16 and the t_{cas1} at least once to a different value, and it could not have been changed back to cb_0 as we know from the C.7.

Notice that we have actually prooved a stronger result above. We have also shown that the string of values written at $idx + 1$ by CAS in line 19 successfully is a subsequence of **all** the transitions of values at idx (not just until t_0).

Lemma C.10 (Valid writes) *Given a FlowPool in a valid state, all writes in all operations produce a FlowPool in a valid state.*

Proof. A new FlowPool is trivially in a valid state.

Otherwise, assume that the FlowPool is in a valid state $\mathbb{S}$. In the rest of the proof, whenever some invariant is trivially unaffected by a write, we omit mentioning it. We start by noting that we already prooved the claim for atomic writes in lines 21, 37 and 75 (which only affect [INV5]) in C.5. We proceed by analyzing each atomic CAS instruction.

CAS in line 44 at time t_1 maintains the invariant [INV1]. This is because its expected value is always $null$, which ensures that the lifecycle of $b.next$ is $null \rightarrow b' : Block$, meaning that the function $reachable(start)$ returns a monotonically growing set. So if $current \in reachable(start)$ at t_0, then this also holds at $t_1 > t_0$. It also maintains [INV2] because the new value nb is always fresh, so $\forall b, b \notin following(b)$. Finally, it maintains [INV3] because it is preceeded with a bounds check and we know from corollary C.6 and the C.3 that all the values in $b.array(idx), idx < LASTELEMPOS$ must be of type $Elem$.

CAS in line 47 at time t_1 maintains the invariant [INV1], since the new value for the $current \neq null$ was read from $b.next$ at $t_0 < t_1$ when the invariant was assumed to hold, and it is still there a t_1, as shown before.

For CAS instructions in lines 20, 113 and 93 that write to index idx we know from C.5 that the value at $idx - 1$ is of type $Elem$. This immediately shows that CAS instructions in lines 113 and 93 maintain [INV3] and [INV4].

For CAS in line 20 we additionally know that it must have been preceeded by a successful CAS in line 19 which previously wrote a $Terminal$ value to $idx + 1$. From C.8 we know that $idx + 1$ is still $Terminal$ when the CAS in line 20 occurs, hence [INV4] is kept.

Finally, CAS in line 19 succeeds only if the value at $idx + 1$ is of type $Terminal$, as shown before in C.8. By the same lemma, the value at idx is either $Terminal$ or $Elem$ at that point, since $idx - 1$ is known to be $Elem$ by C.5. This means that [INV4] is kept.

Lemma C.11 (Housekeeping) *Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, CAS instructions in lines 19, 44 and 47 do not change the abstract pool state $\mathbb{A}$.*

Proof. Since none of the relations *hasElem*, *hasCallback*, *willBeCalled* and *sealedAt* are defined by the value of *current* CAS in line 47 does not change them, hence it does not change the abstract pool state.

No CAS changes the set of scheduled futures, nor is succeeded by a `future` construct so it does not affect the *willBeCalled* relation.

It is easy to see that the CAS in line 44 does not remove any elements, nor make any additional elements reachable, since the new block *nb* which becomes reachable does not contain any elements at that time. Hence the *hasElem* relation is not affected. It does change the value $last(start)$ to nb, but since $nb.array = t \cdot null^{BLOCKSIZE-1}$, where $t \in Terminal$ was previously the last non-null element in $b.array$, it does changes neither the *sealedAt* nor the *hasCallback* relation.

The CAS in line 19 does not make some new element reachable, hence the *hasElem* relation is preserved.

Note now that this CAS does not change the relations *hasCallback* and *sealedAt* as long as there is a value of type $Terminal$ at the preceeding entry idx. We claim that if the CAS succeeds at t_2, then either the value at idx is of type $Terminal$ (trivially) or the CAS did not change the value at $idx + 1$. In other words, if the value at idx at time t_2 is of type $Elem$, then the write by CAS in line 19 does not change the value at $idx + 1$ at t_2. This was, in fact, already shown in the proof of C.9.

The argument above proves directly that relations *hasCallback* and *sealedAt* are not changed by the CAS in line 19.

Lemma C.12 (Append correctness) *Given a FlowPool in state* $\mathbb{S}$ *consistent with some abstract pool state* $\mathbb{A}$*, a successful CAS in line 20 at some time* t_0 *changes the state of the FlowPool to* $\mathbb{S}_0$ *consistent with an abstract pool state* $\mathbb{A}_0$*, such that:*

$\mathbb{A} = (elems, callbacks, seal)$

$\mathbb{A}_0 = (\{elem\} \cup elems, callbacks, seal)$

Furthermore, given a fair scheduler, there exists a time $t_1 > t_0$ *at which the FlowPool is consistent with an abstract pool in state* $\mathbb{A}_1$*, such that:*

$\mathbb{A}_1 = (elems_1, callbacks_1, seal_1)$*, where:*

$\forall (f, called_1) \in callbacks_1, (f, called) \in callbacks \Rightarrow elem \in called_1$

Proof. Assume that the CAS in line 20 succeeds at some time t_3, the CAS in line 19 succeeds at some time $t_2 < t_3$, the read in line 17 occurs at some time $t_1 < t_2$ and the read in line 17 occurs at some time $t_0 < t_1$.

It is easy to see from the invariants, *check* procedure and the corollary C.4 that the CAS in line 20 can only occur if $b = last(start)$.

We claim that for the block $b \in reachable(start)$ such that $b = last(b)$ the following holds at t_2:

$b.array = elem^N \cdot cb_1 \cdot cb_2 \cdot null^{BLOCKSIZE-N-2}$

where $cb_1 = cb_2$, since there was no write to idx after cb_1, otherwise the CAS in line 20 at t_3 would not have been successful (by lemma C.7).

Furthermore, $cb_1 = cb_2$ at t_3, as shown in the C.9. Due to the same lemma, the entries of $b.array$ stay the same until t_3, otherwise the CAS in line 20 would not have been successful. After the successful CAS at t_3, we have:

$b.array = elem^N \cdot e \cdot cb_1 \cdot null^{BLOCKSIZE-N-2}$

where $e : Elem$ is the newly appended element– at t_3 the relation $hasElem(start, e)$ holds, and $sealedAt(start, s)$ and $hasCallback(start, f)$ did not change between t_2 and t_3.

It remains to be shown that $willBeCalled(start, e, f)$ holds at t_3. Given a fair scheduler, within a finite number of steps the future store will contain a request for an asynchronous computation that invokes f on e. The fair scheduler ensures that the future is scheduled within a finite number of steps.

Lemma C.13 (Foreach correctness) *Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, a successful CAS in line 113 at some time t_0 changes the state of the FlowPool to $\mathbb{S}_0$ consistent with an abstract pool state $\mathbb{A}_0$, such that:*

$\mathbb{A} = (elems, callbacks, seal)$

$\mathbb{A}_0 = (elems, (f, \emptyset) \cup callbacks, seal)$

Furthermore, given a fair scheduler, there exists a time $t_1 \geq t_0$ at which the FlowPool is consistent with an abstract pool in state $\mathbb{A}_1$, such that:

$\mathbb{A}_1 = (elems_1, callbacks_1, seal_1)$*, where:*

$elems \subseteq elems_1$

$\forall (f, called_1) \in callbacks_1, elems \subseteq called_1$

Proof. From C.7 and the assumption that the CAS is successful we know that the value at $b.array(idx)$ has not changed between the read in line 106 and the CAS in line 113. From C.5 we know that the value at $idx - 1$ was of type $Elem$ since $b.index$ was read. This means that neither $hasElem(start, e)$ nor $sealedAt$ have changed after the CAS. Since after the CAS there is a $Terminal$ with an additional function f at idx, the $hasCallback(start, f)$ holds after the CAS. Finally, the $willBeCalled(start, e, f)$ holds for all elements e for which the $hasElem(e)$ holds, since the CAS has been preceeded by a call $f(e)$ in line 116 for each element. The C.3 ensures that for each element f was called for stays in the pool indefinitely (i.e. is not removed).

Trivially, the time t_1 from the statement of the lemma is such that $t_1 = t_0$.

Lemma C.14 (Seal correctness) *Given a FlowPool in state $\mathbb{S}$ consistent with some abstract pool state $\mathbb{A}$, a successful CAS in line 93 at some time t_0 changes the state of the FlowPool to $\mathbb{S}_0$ consistent with an abstract pool state $\mathbb{A}_0$, such that:*

$\mathbb{A} = (elems, callbacks, seal)$, *where* $seal \in \{-1\} \cup \{s\}$

$\mathbb{A}_0 = (elems, callbacks, \{s\})$

Proof. Similar to the proof of C.13.

Obstruction-freedom. Given a FlowPool in a valid state, an operation *op* is **obstruction-free** if and only if a thread T executing the operation *op* completes within a finite number of steps given that no other thread was executing the operation *op* since T started executing it.

We say that thread T executes the operation *op* **in isolation**.

Lemma C.15 (Obstruction-free operations) *All operations on FlowPools are obstruction-free.*

Proof. By trivial sequential code analysis supported by the fact that the invariants (especially [INV2]) hold in a valid state.

Safety. From C.11, C.12, C.13 and C.14 directly, along with the fact that all operations executing in isolation complete after a finite number of steps by C.15.

Linearizability. We say that an operation *op* is linearizable if every thread observers that it completes at some time t_0 after it was invoked and before it finished executing.

Theorem C.16 (Linearizable operations) *FlowPool operations* `append` *and* `seal` *are linearizable.*

Linearizable operations. This follows directly from statements about CAS instructions in C.11, C.12 and C.14, along with the fact that a CAS instruction itself is linearizable.

Note that `foreach` starts by executing an asynchronous computation and then returns the control to the caller. This means that the linearization point may happen outside the execution interval of that procedure – so, `foreach` is not linearizable.

D Randomized Batching in the Work-Stealing Tree

This appendix is an extension to the Chapter 5. Unlike the previous two appendices, this appendix does not aim to prove the correctness of the work-stealing tree data structure. Although lock-free, this data structure is a monotonically growing tree, and its correctness is much easier to establish than that of FlowPools or Ctries.

Instead, the goal of this appendix is to present a preliminary analysis of how adding randomization to the batching schedules within single work-stealing nodes impacts scalability. We will theoretically and empirically show that a randomized batching schedule can be beneficial, as long as adequate locality in accessing the data is retained. Note that, in the Chapter 5, we settled for the exponential batching schedule, although this lead to suboptimal speedups for certain workloads. In fact, in Section 5.3, it was shown that for any deterministic batching schedule there exists a workload which results in suboptimal speedup. The appendix assumes that the reader is well acquainted with the content in the Chapter 5, and especially Section 5.3. We start with a theoretical treatment of using a randomized batching schedule, and conclude by studying randomized batching schedules in practice.

Recall that the workload distribution that led to a bad speedup in our evaluation consisted of a sequence of very cheap elements followed by a minority of elements which were computationally very expensive. On the other hand, when we inverted the order of elements, the speedup became linear. The exponential backoff approach is designed to start with smaller batches first in hopes of hitting the part of the workload which contains most work as early as possible. This allow other workers to steal larger pieces of the remaining work, hence allowing a more fine grained batch subdivision. In this way the scheduling algorithm is workload-driven – it gives itself its own feedback. In the absence of other information about the workload, the knowledge that some worker is processing some part of the workload long enough that it can be stolen from is the best sign that the workload is different than the baseline, and that the batch subdivision can circumvent the baseline constraint. This heuristic worked in the example from figure 5.14-36 when the expensive elements were reached first, but failed when they were reached in the last, largest batch, and we know that there has to be a largest batch by Lemma 5.1 – a single worker must divide the range into batches the mean size of which has a lower bound. In fact, no other deterministic scheduler can yield an optimal speedup for all schedules, as shown by Lemma 5.2. For this reason we look into randomized schedulers.

In particular, in the example from the evaluation we would like the scheduler to put the smallest batches at the end of the range, but we have no way of knowing if the most expensive elements are positioned somewhere else. With this in mind we randomize the batching order. The baseline constraint still applies in oblivious conditions, so we have to pick different batch sizes with respect to the constraints from Lemma 5.1. Lets pick exactly the same set of exponentially increasing batches, but place consequent elements into different batches randomly. In other words, we permute the elements of the range and then apply the previous scheme. We expect some of the more expensive elements to be assigned to the smaller batches, giving other workers a higher opportunity to steal a part of the work.

In evaluating the effectiveness of this randomized approach we will assume a particular distribution we found troublesome. We define it more formally.

Step workload distribution. A *step workload distribution* is a function which assigns a computational cost $w(i)$ to each element i of the range of size N as follows:

$$w(i) = \begin{cases} w_e, & i \in [i_1, i_2] \\ w_0, & i \notin [i_1, i_2] \end{cases} \tag{D.1}$$

where $[i_1, i_2]$ is a subsequence of the range, w_0 is the minimum cost of computation per element and $w_e \gg w_0$. If $w_e \geq f \cdot T_d$, where f is the computation speed and T_d is the worker delay, then we additionally call the workload *highly irregular*. We call $D = 2^d = i_2 - i_1$ the *span* of the step distribution. If $(N - D) \cdot \frac{w_0}{f} \leq T_d$ we also call the workload *short*.

We can now state the following lemma. We will refer to the randomized batching schedule we have described before as the **randomized permutation with an exponential backoff**. Note that we implicitly assume that the worker delay T_d is significantly greater than the time T_c spent scheduling a single batch (this was certainly true in our experimental evaluation).

Lemma D.1 *When parallelizing a workload with a highly irregular short step workload distribution the expected speedup inverse of a scheduler using randomized permutations* *with an exponential backoff is:*

$$\langle s_p^{-1} \rangle = \frac{1}{P} + (1 - \frac{1}{P}) \cdot \frac{(2^k - 2^d - 1)!}{(2^k - 1)!} \cdot \sum_{i=0}^{k-1} 2^i \frac{(2^k - 2^i - 1)!}{(2^k - 2^i - 2^d)!} \tag{D.2}$$

where $D = 2^d \gg P$ is the span of the step workload distribution.

Proof. The speedup s_p is defined as $s_p = \frac{T_0}{T_p}$ where T_0 is the running time of the optimal sequential execution and T_p is the running time of the parallelized execution. We implicitly assume that all processors have the same the same computation speed f. Since $w_e \gg w_0$, the total amount of work that a sequential loop executes is arbitrarily close to $D \cdot w_e$, so $T_0 = \frac{D}{f}$. When we analyze the parallel execution, we will also ignore the work w_0. We will call the elements with cost w_e *expensive*.

We assumed that the workload distribution is highly irregular. This means that if the first worker ω starts the work on an element from $[i_1, i_2]$ at some time t_0 then at the time $t_1 = t_0 + \frac{w_e}{f}$ some other worker must have already started working as well, because $t_1 - t_0 \geq T_d$. Also, we have assumed that the workload distribution is short. This means that the first worker ω can complete work on all the elements outside the interval $[i_1, i_2]$ before another worker arrives. Combining these observations, as soon as the first worker arrives at an expensive element, it is possible for the other workers to parallelize the rest of the work.

We assume that after the other workers arrive there are enough elements left to efficiently parallelize work on them. In fact, at this point the scheduler will typically change the initially decided batching schedule – additionally arriving workers will steal and induce a more fine-grained subdivision. Note, however, that the other workers cannot subdivide the batch on which the current worker is currently working on – that one is no longer available to them. The only batches with elements of cost w_e that they can still subdivide are the ones coming after the first batch in which the first worker ω found an expensive element. We denote this batch with c_ω. The batch c_ω may, however, contain additional expensive elements and the bigger the batch the more probable this is. We will say that the total number of expensive elements in c_ω is X. Finally, note that we assumed that $D \gg P$, so our expression will only be an approximation if D is very close to P.

We thus arrive at the following expression for speedup:

$$s_p = \frac{D}{X + \frac{D-X}{P}} \tag{D.3}$$

Speedup depends on the value X. But since the initial batching schedule is random, the speedup depends on the random variable and is itself random. For this reason we will look for its expected value. We start by finding the expectation of the random variable X.

We will now solve a more general problem of placing balls to an ordered set of bins and apply the solution to finding the expectation of X. There are k bins, numbered from 0 to $k-1$. Let c_i denote the number of balls that fit into the ith bin. We randomly assign D balls to bins, so that the number of balls in each bin i is less than or equal to c_i. In other words, we randomly select D slots from all the $N = \sum_{i=0}^{k-1} c_i$ slots in all the bins together. We then define the random variable X to be the number of balls in the non-empty bin with the smallest index i. The formulated problem corresponds to the previous one – the balls are the expensive elements and the bins are the batches.

An alternative way to define X is as follows [Makholm(2013)]:

$$X = \sum_{i=0}^{k-1} \begin{cases} \text{no. balls in bin } i & \text{if all the bins } j < i \text{ are empty} \\ 0 & \text{otherwise} \end{cases} \tag{D.4}$$

Applying the linearity property, the expectation $\langle X \rangle$ is then:

$$\langle X \rangle = \sum_{i=0}^{k-1} \langle \text{no. balls in bin } i \text{ if the bins } j < i \text{ are empty; 0 otherwise} \rangle \tag{D.5}$$

The expectation in the sum is conditional on the event that all the bins coming before i are empty. We call the probability of this event p_i. We define b_i as the number of balls in any bin i. From the properties of conditional expectation we than have:

$$\langle X \rangle = \sum_{i=0}^{k-1} p_i \cdot \langle b_i \rangle \tag{D.6}$$

The number of balls in any bin is the sum of the balls in all the slots of that bin which spans slots n_{i-1} through $n_{i-1} + c_i$. The expected number of balls in a bin i is thus:

$$\langle b_i \rangle = \sum_{i=n_{i-1}}^{n_{i-1}+c_i} \langle \text{expected no. balls in a single slot} \rangle \tag{D.7}$$

We denote the total capacity of all the bins $j \geq i$ as q_i (so that $q_0 = N$ and $q_{k-1} = 2^{k-1}$). We assign balls to slots randomly with a uniform distribution – each slot has a probability $\frac{D}{q_i}$ of being selected. Note that the denominator is not N – we are calculating a conditional probability for which all the slots before the ith bin are empty. The expected number of balls in a single slot is thus $\frac{D}{q_i}$. It follows that:

$$\langle b_i \rangle = c_i \cdot \frac{D}{q_i} \tag{D.8}$$

Next, we compute the probability p_i that all the bins before the bin i are empty. We do this by counting the events in which this is true, namely, the number of ways to assign balls in bins $j \geq i$. We will pick combinations of D slots, one for each ball, from a set of q_i slots. We do the same to enumerate all the assignments of balls to bins, but with $N = q_0$ slots, and obtain:

$$p_i = \frac{\binom{q_i}{D}}{\binom{q_0}{D}} \tag{D.9}$$

We assumed here that $q_i \geq D$, otherwise we cannot fill all D balls into bins. We could create a constraint that the last batch is always larger than the number of balls. Instead, we simply define $\binom{q_i}{D} = 0$ if $q_i < D$ – there is no chance we can fit more than q_i balls to q_i slots. Combining these relations, we get the following expression for $\langle X \rangle$:

$$\langle X \rangle = D \cdot \frac{(q_0 - D)!}{q_0!} \sum_{i=0}^{k-1} c_i \cdot \frac{(q_i - 1)!}{(q_i - D)!} \tag{D.10}$$

We use this expression to compute the expected speedup inverse. By the linearity of expectation:

$$\langle s_p^{-1} \rangle = \frac{1}{P} + \left(1 - \frac{1}{P}\right) \cdot \frac{(q_0 - D)!}{q_0!} \sum_{i=0}^{k-1} c_i \cdot \frac{(q_i - 1)!}{(q_i - D)!} \tag{D.11}$$

This is a more general expression than the one in the claim. When we plug in the exponential backoff batching schedule, i.e. $c_i = 2^i$ and $q_i = 2^k - 2^i$, the lemma follows.

The expression derived for the inverse speedup does not have a neat analytical form, but we can evaluate it for different values of d to obtain a diagram. As a sanity check, the worst expected speedup comes with $d = 0$. If there is only a single expensive element in the range, then there is no way to parallelize execution – the expression gives us the speedup 1. We expect a better speedup as d grows – when there are more expensive elements, it is easier for the scheduler to stumble upon some of them. In fact, for $d = k$, with the conventions established in the proof, we get that the speedup is $\frac{1}{P} + \left(1 - \frac{1}{P}\right) \cdot \frac{c_0}{D}$.

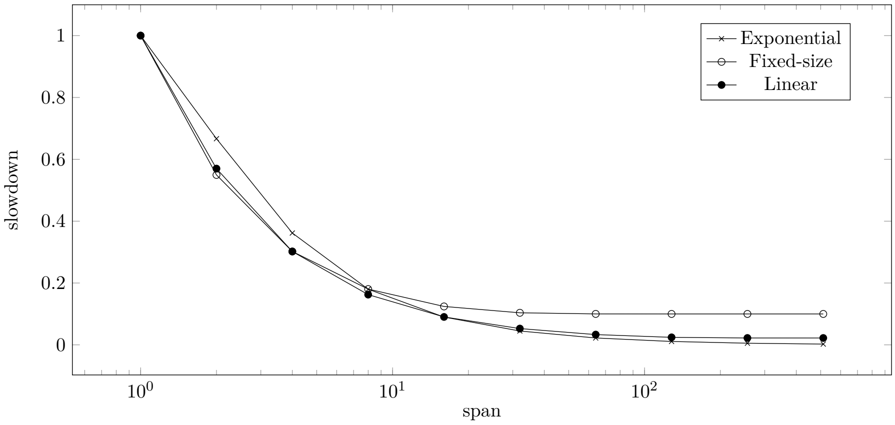


Figure D.1: Randomized Scheduler Executing Step Workload – Speedup vs. Span

This means that when all the elements are expensive the proximity to the optimal speedup depends on the size c_0 of the first batch – the less elements in it, the better. Together with the fact that many applications have uniform workloads, this is also the reason why we advocate exponential backoff for which the size of the first batch is 1.

We call the term $\frac{(q_0-D)!}{q_0!}\sum_{i=0}^{k-1} c_i \cdot \frac{(q_i-1)!}{(q_i-D)!}$ the *slowdown* and plot it with respect to span D on the diagram in Figure D.1. In this diagram we choose $k = 10$, and the number of elements $N = 2^{10} = 1024$. As the term nears 1, the speedup nears 1. As the term approaches 0, the speedup approaches the optimal speedup P. The quicker the term approaches 0 as we increase d, the better the scheduler. We can see that fixed-size batching should work better than the exponential backoff if the span D is below 10 elements, but is much worse than the exponential backoff otherwise. Linearly increasing the batch size from 0 in some step $a = \frac{2\cdot(2^k-1)}{k\cdot(k-1)}$ seems to work well even for span $D < 10$. However, the mean batch size $\overline{c_i} = \frac{S}{k}$ means that this approach may easily violate the baseline constraint, and for $P \approx D$ the formula is an approximation anyway.

The conclusion is that selecting a random permutation of the elements should work very well in theory. For example, the average speedup becomes very close to optimal if less than $D = 10$ elements out of $N = 1024$ are expensive. However, randomly permuting elements would in practice either require a preparatory pass in which the elements are randomly copied or would require the workers to randomly jump through the array, leading to cache miss issues. In both cases the baseline performance would be violated. Even permuting the order of the batches seems problematic, as it would require storing information about where each batch started and left off, as well as its intermediate result.

There are many approaches we could study, many of which could have viable implementations, but we focus on a particular one which seems easy to implement for ranges and other data structures. Recall that in the example in Figure 5.14-36 the interval with expensive elements was positioned at the end of the range. What if the worker alternated

```
def workOn(ptr: Ptr): Boolean =
  val node = READ(ptr.child)
  var batch = -1
  var sum = 0
  do
    val p = READ(node.progress)
    if (notCompleted(p) && notStolen(p))
      if (coinToss())
        batchs = tryAdvanceLeft(node, p)
        if (notStolen(batch)) sum += kernel(p, p + decodeStep(batch))
      else
        batch = tryAdvanceRight(node, p)
        if (notStolen(batch)) sum += kernel(p, p + decodeStep(batch))
    else batch = -1
  while (batch ≠ -1)
  complete(sum, ptr)
```

Figure D.2: Randomized `loop` Method

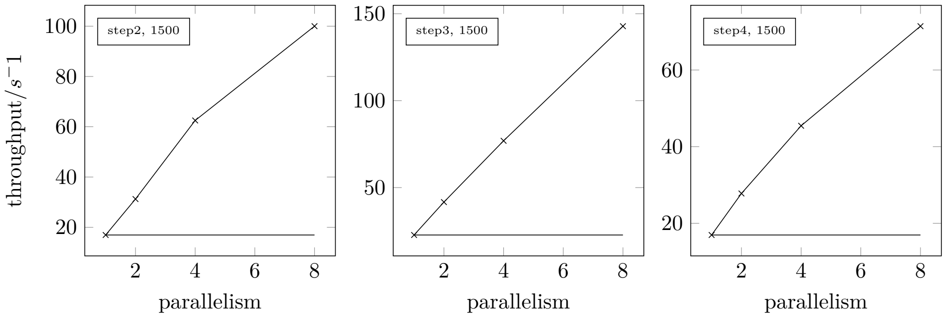


Figure D.3: The Randomized Work-Stealing Tree and the STEP3 Workload

the batch in each step by tossing the coin to decide if the next batch should be from the left (start) or from the right (end)? Then the worker could arrive at the expensive interval on the end while the batch size is still small with a relatively high probability. The changes to the work-stealing tree algorithm are minimal – in addition to another field called `rresult` (the name of which should shed some light on the previous choice of name for `lresult`), we have to modify the `workOn`, `complete` and `pushUp` methods. While the latter two are straightforward, the lines 41 through 45 of `workOn` are modified. The new `workOn` method is shown in Figure D.2.

The main issue here is to encode and atomically update the iteration state, since it consists of two pieces of information – the left and the right position in the subrange. We can encode these two positions by using a long integer field and a long CAS operation to update it. The initial 32 bits can contain the position on the left side of the subrange and the subsequent 32 on the right side. With this in mind, the methods `tryAdvanceLeft`, `tryAdvanceRight`, `notStolen`, `notCompleted` and `decodeStep` should be straightforward.

We evaluate the new scheduler on the distribution from Figure 5.14-36 and show the results in Figure D.3. The first two diagrams (STEP2 and STEP3) show that with the expensive interval at the beginning and the end of the range the work-stealing tree achieves a close to optimal speedup. However, there is still a worst case scenario that we have to consider, and that is to have a step workload with the expensive interval exactly in the middle of the range. Intuition tells us that the probability to hit this interval early on is smaller, since a worker has to progress through more batches to arrive at it. The workload STEP4 in the third diagram of Figure D.3 contains around 25% expensive elements positioned in the middle of the range. The speedup is decent, but not linear for STEP4, since the bigger batches seem to on average hit the middle of the range more often.

These preliminary findings indicate that randomization can help scheduling both in theory and in practice. We conclude that the problem of overcoming particularly bad workload distributions is an algorithmic problem of finding a randomized batching schedule which can be computed and maintained relatively quickly. Here, an important concern is that of data locality – although a randomized batching schedule that picks elements at random can cope with irregularity well, it slows down the sequential baseline due to cache misses. We do not dive into this problem further, but leave it as part of future research.

Bibliography

[Adelson-Velsky and Landis(1962)] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146: 263–266, 1962.

[Agesen et al.(2000)Agesen, Detlefs, Flood, Garthwaite, Martin, Shavit, and Jr.] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Dcas-based concurrent deques, 2000.

[Allen et al.(2007)Allen, Chase, Hallett, Luchangco, Maessen, Ryu, Jr., and Tobin] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007. URL `http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`.

[Areias and Rocha(2014)] M. Areias and R. Rocha. A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs. In C. Grelck, editor, *7th International Symposium on High-level Parallel Programming and Applications (HLPP 2014)*, pages 259–278, Amsterdam, Netherlands, July 2014.

[Arora et al.(1998)Arora, Blumofe, and Plaxton] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM. ISBN 0-89791-989-0. doi: 10.1145/277651.277678. URL `http://doi.acm.org/10.1145/277651.277678`.

[Arvind et al.(1989)Arvind, Nikhil, and Pingali] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Prog. Lang. and Sys.*, 11(4):598–632, October 1989.

[Bagwell(2001)] Phil Bagwell. Ideal hash trees, 2001.

[Bagwell(2002)] Phil Bagwell. Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays. Technical report, 2002.

[Bagwell and Rompf(2011)] Philip Bagwell and Tiark Rompf. RRB-Trees: Efficient Immutable Vectors. Technical report, 2011.

[Baskins(2000)] Douglas Baskins. Judy array implementation. http://judy.sourceforge.net/, 2000.

[Blelloch(1992)] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[Blumofe and Leiserson(1999)] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL `http://doi.acm.org/10.1145/324133.324234`.

[Blumofe et al.(1995)Blumofe, Joerg, Kuszmaul, Leiserson, Randall, and Zhou] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[Boehm et al.(1995)Boehm, Atkinson, and Plass] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Softw. Pract. Exper.*, 25(12): 1315–1330, December 1995. ISSN 0038-0644. doi: 10.1002/spe.4380251203. URL `http://dx.doi.org/10.1002/spe.4380251203`.

[Bronson(2011a)] Nathan Bronson. *Composable Operations on High-Performance Concurrent Collections.* PhD thesis, 2011a.

[Bronson(2011b)] Nathan Bronson. Scalastm implementation. https://nbronson.github.io/scala-stm/, 2011b.

[Bronson et al.(2010a)Bronson, Casper, Chafi, and Olukotun] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010a. ISSN 0362-1340. doi: 10.1145/1837853.1693488. URL `http://doi.acm.org/10.1145/1837853.1693488`.

[Bronson et al.(2010b)Bronson, Chafi, and Olukotun] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. Ccstm: A library-based stm for scala. In *In The First Annual Scala Workshop at Scala Days*, 2010b.

[Burke et al.(2011)Burke, Knobe, Newton, and Sarkar] Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. Concurrent collections programming model. In *Encyclopedia of Parallel Computing*, pages 364–371. 2011.

[Burmako and Odersky(2012)] Eugene Burmako and Martin Odersky. Scala Macros, a Technical Report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012. URL `http://scalamacros.org/`.

[Buss et al.(2010)Buss, Harshvardhan, Papadopoulos, Amato, and Rauchwerger] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-908-4. doi: 10.1145/1815695.1815713. URL `http://doi.acm.org/10.1145/1815695.1815713`.

[Chamberlain(2013)] Bradford L. Chamberlain. A brief overview of Chapel, 2013.

[Charles et al.(2005)Charles, Grothoff, Saraswat, von Praun, and Sarkar] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852. URL `http://doi.acm.org/10.1145/1094811.1094852`.

[Chase and Lev(2005)] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: 10.1145/1073970.1073974. URL `http://doi.acm.org/10.1145/1073970.1073974`.

[Click(2007)] Cliff Click. Towards a scalable non-blocking coding style, 2007. URL `http://www.azulsystems.com/events/javaone\_2007/2007\_LockFreeHash.pdf`.

[Cong et al.(2008)Cong, Kodali, Krishnamoorthy, Lea, Saraswat, and Wen] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

[Cormen et al.(2001)Cormen, Leiserson, Rivest, and Stein] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[De La Briandais(1959)] Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. ACM. doi: 10.1145/1457838.1457895. URL `http://doi.acm.org/10.1145/1457838.1457895`.

[Dragos(2010)] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.

[Dragos and Odersky(2009)] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565830. URL `http://doi.acm.org/10.1145/1565824.1565830`.

[Ellen et al.(2010)Ellen, Fatourou, Ruppert, and van Breugel] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835736. URL `http://doi.acm.org/10.1145/1835698.1835736`.

[Fich et al.(2004)Fich, Hendler, and Shavit] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 80–87, New York, NY, USA, 2004. ACM. ISBN 1-58113-802-4. doi: 10.1145/1011767.1011780. URL `http://doi.acm.org/10.1145/1011767.1011780`.

[Fredkin(1960)] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960. ISSN 0001-0782. doi: 10.1145/367390.367400. URL `http://doi.acm.org/10.1145/367390.367400`.

[Friedman and Wise(1976)] Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, 1976.

[Frigo et al.(1998)Frigo, Leiserson, and Randall] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277725. URL `http://doi.acm.org/10.1145/277650.277725`.

[Georges et al.(2007)Georges, Buytaert, and Eeckhout] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

[Goetz et al.(2006)Goetz, Bloch, Bowbeer, Lea, Holmes, and Peierls] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006. ISBN 0321349601. URL `http://www.amazon.de/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601`.

[Haller et al.(2012)Haller, Prokopec, Miller, Klang, Kuhn, and Jovanovic] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Scala improvement proposal: Futures and promises (SIP-14). 2012. See `http://docs.scala-lang.org/sips/pending/futures-promises.html`.

[Halstead(1985)] Jr. R. H. Halstead. MultiLISP: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. and Sys.*, 7(4):501–538, October 1985.

[Harris(2001)] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42605-1. URL `http://dl.acm.org/citation.cfm?id=645958.676105`.

[Harris et al.(2002)Harris, Fraser, and Pratt] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00073-9. URL `http://dl.acm.org/citation.cfm?id=645959.676137`.

[Henry C. Baker and Hewitt(1977)] Jr. Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proc. Symp. on Art. Int. and Prog. Lang.*, 1977.

[Herlihy(1993)] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993. ISSN 0164-0925. doi: 10.1145/161468.161469. URL `http://doi.acm.org/10.1145/161468.161469`.

[Herlihy and Moss(1993)] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. URL `http://doi.acm.org/10.1145/165123.165164`.

[Herlihy and Shavit(2008)] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[Herlihy and Wing(1990)] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Hinze and Paterson(2006)] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, March 2006. ISSN 0956-7968. doi: 10.1017/S0956796805005769. URL `http://dx.doi.org/10.1017/S0956796805005769`.

[Hummel et al.(1992)Hummel, Schonberg, and Flynn] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, August 1992. ISSN 0001-0782. doi: 10.1145/135226.135232. URL `http://doi.acm.org/10.1145/135226.135232`.

[Iverson(1962)] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[Jones and Lins(1996)] Richard E. Jones and Rafael Dueire Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996. ISBN 0-471-94148-4.

[Kaplan and Tarjan(1995)] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 93–102, New York, NY, USA, 1995. ACM. ISBN 0-89791-718-9. doi: 10.1145/225058.225090. URL `http://doi.acm.org/10.1145/225058.225090`.

[Knight(1986)] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319854. URL `http://doi.acm.org/10.1145/319838.319854`.

[Koelbel and Mehrotra(1991)] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):440 –451, oct 1991. ISSN 1045-9219. doi: 10.1109/71.97901.

[Kruskal and Weiss(1985)] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, October 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.231547. URL `http://dx.doi.org/10.1109/TSE.1985.231547`.

[Kung and Lehman(1980)] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, September 1980. ISSN 0362-5915. doi: 10.1145/320613.320619. URL `http://doi.acm.org/10.1145/320613.320619`.

[Lea(2000)] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337465. URL `http://doi.acm.org/10.1145/337449.337465`.

[Lea(2014)] Doug Lea. Doug lea's workstation, 2014. URL `http://g.oswego.edu/`.

[Lentz(2013)] Dan Lentz. Common lisp ctrie implementation, 2013. URL `https://github.com/danlentz/cl-ctrie`.

[Levenstein(2012)] Roman Levenstein. Java ctrie implementation, 2012. URL `https://github.com/romix/java-concurrent-hash-trie-map`.

[Litwin et al.(1989)Litwin, Sagiv, and Vidyasankar] W. Litwin, Y. Sagiv, and K. Vidyasankar. Concurrency and trie hashing. *Acta Inf.*, 26(7):597–614, September 1989. ISSN 0001-5903. doi: 10.1007/BF00288973. URL `http://dx.doi.org/10.1007/BF00288973`.

[Litwin(1981)] Witold Litwin. Trie hashing. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 19–29, New York, NY, USA, 1981. ACM. ISBN 0-89791-040-0. doi: 10.1145/582318.582322. URL `http://doi.acm.org/10.1145/582318.582322`.

[Makholm(2013)] Henning Makholm. Answer at the mathematics forum (stackexchange). https://math.stackexchange.com/questions/297375/bins-in-balls-where-bin-size-grows-exponentially, 2013.

[Meijer(2012)] Erik Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[Mellor-Crummey(1987)] John M. Mellor-Crummey. Concurrent queues: Practical fetch-and-Φ algorithms. 1987.

[Michael(2002)] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7. doi: 10.1145/564870.564881. URL `http://doi.acm.org/10.1145/564870.564881`.

[Michael(2004)] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. ISSN 1045-9219. doi: 10.1109/TPDS.2004.8. URL `http://dx.doi.org/10.1109/TPDS.2004.8`.

[Michael and Scott(1996)] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[Moir and Shavit(2004)] Mark Moir and Nir Shavit. Concurrent data structures, 2004.

[Odersky(2009)] Martin Odersky. Scala 2.8 collections. Technical report, EPFL, Lausanne, November 2009.

[Odersky and Moors(2009)] Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In Ravi Kannan and K Narayan

Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-13-2. doi: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2009.2338. URL `http://drops.dagstuhl.de/opus/volltexte/2009/2338`.

[Okasaki(1996)] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1996. AAI9813847.

[Okasaki(1997)] Chris Okasaki. Catenable double-ended queues. In *In Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 66–74. ACM Press, 1997.

[Okasaki(1998)] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

[Peyton Jones(2008)] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_10. URL `http://dx.doi.org/10.1007/978-3-540-89330-1_10`.

[Polychronopoulos and Kuck(1987)] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, December 1987. ISSN 0018-9340. doi: 10.1109/TC.1987.5009495. URL `http://dx.doi.org/10.1109/TC.1987.5009495`.

[Prokopec(2014a)] Aleksandar Prokopec. https://github.com/storm-enroute/reactive-collections (Reactive Collections repository), 2014a.

[Prokopec(2014b)] Aleksandar Prokopec. Scalameter website, 2014b. URL `http://scalameter.github.io`.

[Prokopec and Petrashko(2013)] Aleksandar Prokopec and Dmitry Petrashko. Scalablitz documentation, 2013. URL `http://scala-blitz.github.io/home/documentation/`.

[Prokopec et al.(2011a)Prokopec, Bagwell, and Odersky] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. Cache-Aware Lock-Free Concurrent Hash Tries. Technical report, 2011a.

[Prokopec et al.(2011b)Prokopec, Bagwell, and Odersky] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. Lock-free resizeable concurrent tries. In *LCPC*, pages 156–170, 2011b.

[Prokopec et al.(2011c)Prokopec, Bagwell, Rompf, and Odersky] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 136–147, Berlin, Heidelberg, 2011c. Springer-Verlag. ISBN 978-3-642-23396-8. URL `http://dl.acm.org/citation.cfm?id=2033408.2033425`.

[Prokopec et al.(2012a)Prokopec, Bronson, Bagwell, and Odersky] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. pages 151–160, 2012a. doi: 10.1145/2145816.2145836. URL `http://doi.acm.org/10.1145/2145816.2145836`.

[Prokopec et al.(2012b)Prokopec, Miller, Schlatter, Haller, and Odersky] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012b.

[Prokopec et al.(2014a)Prokopec, Haller, and Odersky] Aleksandar Prokopec, Philipp Haller, and Martin Odersky. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Scala 2014*, 2014a. URL `https://github.com/storm-enroute/reactive-collections,http://reactive-collections.com/`.

[Prokopec et al.(2014b)Prokopec, Petrashko, and Odersky] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. On Lock-Free Work-stealing Iterators for Parallel Data Structures. Technical report, 2014b.

[Pugh(1990a)] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990a. ISSN 0001-0782. doi: 10.1145/78973.78977. URL `http://doi.acm.org/10.1145/78973.78977`.

[Pugh(1990b)] William Pugh. Concurrent maintenance of skip lists. Technical report, College Park, MD, USA, 1990b.

[Reinders(2007)] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[Roy and Haridi(2004)] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN 0-262-22069-5.

[Sarkar(2000)] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 153–166, New York, NY, USA, 2000. ACM. ISBN 1-58113-270-0. doi: 10.1145/335231.335246. URL `http://doi.acm.org/10.1145/335231.335246`.

[Scherer et al.(2009)Scherer, Lea, and Scott] William N. Scherer, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.

[Schlatter et al.(2012)Schlatter, Prokopec, Miller, Haller, and Odersky] Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. Multi-lane flowpools: A detailed look. Technical report, EPFL, Lausanne, September 2012.

[Schlatter et al.(2013)Schlatter, Prokopec, Miller, Haller, and Odersky] Tobias Schlatter, Aleksandar Prokopec, Heather Miller, Philipp Haller, and Martin Odersky. Flowseqs: Barrier-free parseqs. Technical report, EPFL, Lausanne, January 2013.

[Schröder(2014)] Michael Schröder. Haskell ctrie implementation, 2014. URL `http://hackage.haskell.org/package/ctrie`.

[Shalev and Shavit(2006)] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006. ISSN 0004-5411. doi: 10.1145/1147954.1147958. URL `http://doi.acm.org/10.1145/1147954.1147958`.

[Shavit and Touitou(1995)] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224987. URL `http://doi.acm.org/10.1145/224964.224987`.

[Steele(2009)] Guy Steele. Organizing functional code for parallel execution; or, foldl and foldr considered slightly harmful. International Conference on Functional Programming (ICFP), 2009.

[Steele(2010)] Guy Steele. How to think about parallel programming: Not! Strange Loop Conference, 2010.

[Sujeeth(2013)] Arvind K. Sujeeth. Composition and reuse with compiled domain-specific languages. In *In Proceedings of ECOOP*, 2013.

[Tardieu et al.(2012)Tardieu, Wang, and Lin] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145850. URL `http://doi.acm.org/10.1145/2145816.2145850`.

[Toub(2010)] Stephen Toub. Patterns of parallel programming, 2010.

[Tzen and Ni(1993)] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4

(1):87–98, January 1993. ISSN 1045-9219. doi: 10.1109/71.205655. URL `http://dx.doi.org/10.1109/71.205655`.

[Y. Lev and Shavit(2006)] V. Luchangco Y. Lev, M. Herlihy and N. Shavit. A provably correct scalable skiplist (brief announcement). In *Proc. of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006)*, 2006.

Aleksandar Prokopec

Chemin de la Prairie 62
1007 Lausanne
+41 78 948 93 47
aleksandar.prokopec@gmail.com

Strengths:

- Concurrent and parallel programming specialist
- 8 years of practical experience in programming
- Strong analytical skills

Professional Experience

2009-2013 Scala Development Team, EPFL

- Implemented the Parallel Collections data-parallel framework in the standard library that outperforms standard library collections by a factor of 2-7x
- Co-authored the Futures and Promises standard library module forasynchronous programming
- Maintained and fixed bugs in the Scala standard library and parts of the compiler

2009-2013 Doctoral Assistant, LAMP, EPFL

- Designed lock-free concurrent data-structures with better scalability than JDK concurrent collections
- Authored research papers that appeared at the top-tier conferences (publications list: http://axel22.github.io/home/professional/)
- Participated in teaching activities, gave talks at developer and academic conferences

Education

2009-2014 PhD in Computer Science (expected in 2014), EPFL
2004-2009 M.A. in Computer Science, Faculty of Electrical Engineering and Computing, Zagreb

Technical Skills

Concurrent and Parallel Programming
Lock-free algorithms and data structures, data-parallel scheduling algorithms, parallel programming runtimes
Data-structures
Worked with and designed concurrent, mutable and immutable, reactive data-structures
Programming Languages, Frameworks, Tools
Java, Scala, C/C++, C#, OpenGL, GLSL, JavaScript, HTML, CSS, MPI, CUDA, ARM assembler

Projects

Scala Parallel Collections (PhD thesis)–I work on a data-parallel programming framework for the Scala programming language that aims to deliver efficient shared-memory parallel programming to the JVM. My main goals were to eliminate performance penalties related to generic programming on the JVM and deliver a scalable data-parallel runtime. During my thesis I developed a runtime scheduling system that achieves up to 3x better load-balancing for irregular workloads and a framework that achieves 50-100x better baseline performance on the JVM through compiler optimizations.

Evolutionary Computing IDE (graduation thesis)– Java-based IDE for algorithm development, visualization and performance tuning, with a focus on evolutionary algorithms. (https://github.com/axel22/Evolutionary-Computing-IDE).

ScalaMeter – microbenchmarking and performance regression testing framework in the style of Google Caliper, focused on accurately measuring performance on the JVM and producing reports (http://axel22.github.io/scalameter).

VHDLLab – online Java-based IDE for digital circuit modeling and simulation, used in introductory Digital Electronics courses at the Faculty of Electrical Engineering in Zagreb (https://github.com/mbezjak/vhdllab).

Awards and Honors

IC Best Teaching Assistant Award, EPFL 2013.
Best Student Presentation Award, LCPC 2011.
University of Zagreb Rector Award for the Best Student Project (VHDLLab), 2008.
Faculty of Electrical Engineering and Computing "Josip Loncar" Award for Excellent Students 2007.
Faculty of Electrical Engineering and Computing "Josip Loncar" Award for Excellent Students 2006.
Croatian National Scholarship for Excellent Students, 2005.
International Physics Olympiad, IPhO 2004.
State Physics Competition 1st place, 2004.

Languages

Croatian	Native language
English	C2, used at work for 4 years, published research papers in English
German	B2
French	A2

Personal Information

27 years old, single, Slovenian citizenship, Swiss permit B
Hobbies: sports (running, hiking and cycling), developing an open source computer game engine using OpenGL and Scala

Aleksandar Prokopec

Chemin de la Prairie 62
1007 Lausanne
+41 78 948 93 47
aleksandar.prokopec@gmail.com

Strengths:

- Expert für paralleleProgrammierung
- 8 JahrepraktischeErfahrung in der Programmierung
- starkeanalytischeFähigkeiten

Berufserfahrung

2009-2013 Scala Development Team, EPFL

- Implementiert die Parallele Collections in der ScalaStandard-Bibliothek, die Standard-BibliothekCollectionsübertrifftmiteinemFaktor von 2-7x
- Co-Autor die Futures und Promises Standard-Bibliothek-Modulfür die asynchroneProgrammierung
- Bug-fixing in der Scala-Standardbibliothek und Teile des ScalaCompilers

2009-2013 Doktorassistent, LAMP, EPFL

- Entwickelt Lock-freien Datenstrukturenmiteiner besseren Skalierbarkeitals JDK Collections
- SchriebForschungsarbeiten, die an den Top-Tier-Konferenzenerschienen (Publikationsliste: http://axel22.github.io/home/professional/)
- eilnahme in der Lehre, gab Vorträge auf wissenschaftlichen Konferenzen

Bildung

2009-2014 PhDInformatik, EPFL
2004-2009 M.A. Informatik, Faculty of Electrical Engineering and Computing, Zagreb

TechnischeFähigkeiten

Concurrent und paralleleProgrammierung
Lock-freienAlgorithmen und Datenstrukturen, Daten-Parallel-Scheduling-Algorithmen
Datenstrukturen
Gearbeitet und entwickeltconcurrent, veränderliche und unveränderlicheDatenstrukturen
Programmiersprachen, Frameworks, Werkzeuge
Java, Scala, C/C++, C#, OpenGL, GLSL, JavaScript, HTML, CSS, MPI, CUDA, ARM assembler

Projekte

Scala Parallel Collections (PhD thesis)–Icharbeitete auf Parallel CollectionsModulfür die Scala-Programmiersprache, die eineeffizienteparallele Shared-Memory-Programmierungzu der JVM bringt. MeineHauptzielewaren, um LeistungseinbußenzugenerischenProgrammierung auf der JVM Zusammenhangzubeseitigen und lieferneineskalierbareDaten-Parallel-Laufzeit.WährendmeinerThesishabeicheineLaufzeit-Scheduling-System erreicht, die biszu 3x schnellerfürunregelmäßigeArbeitsbelastung (http://docs.scala-lang.org/overviews/parallel-collections/overview.html).

Evolutionary Computing IDE (M.A. thesis)– Java-basierte IDE für die Entwicklung von Algorithmen, Visualisierungs-und Performance-Tuning, miteinemSchwerpunkt auf evolutionärenAlgorithmen (https://github.com/axel22/Evolutionary-Computing-IDE).

ScalaMeter – microbenchmarking und LeistungRegressionstestsModul for JVM (http://axel22.github.io/scalameter).

Awards and Honors

IC Best Teaching Assistant Award, EPFL 2013.
Best Student Presentation Award, LCPC 2011.
University of Zagreb Rector Award for the Best Student Project (VHDLLab), 2008.
Faculty of Electrical Engineering and Computing “Josip Loncar” Award for Excellent Students 2007.
Faculty of Electrical Engineering and Computing “Josip Loncar” Award for Excellent Students 2006.
Croatian National Scholarship for Excellent Students, 2005.
International Physics Olympiad, IPhO 2004.
State Physics Competition 1st place, 2004.

Sprache

Kroatisch	Müttersprache
Englisch	C2, nutzt auf der Arbeit5-Jahrelang, schriebForschungsarbeitenin Englisch
Deutsch	B2
Französich	A2

PersönlicheInformationen

27Jahre alt, unverheiratet, SlowenischStaatsbürgerschaft, Schweizer Permit B
Hobbies: Sporte (laufen, steigen and cycling), Entwicklungeines Open-Source-Computer-Spiel-Engine mitOpenGL und Scala