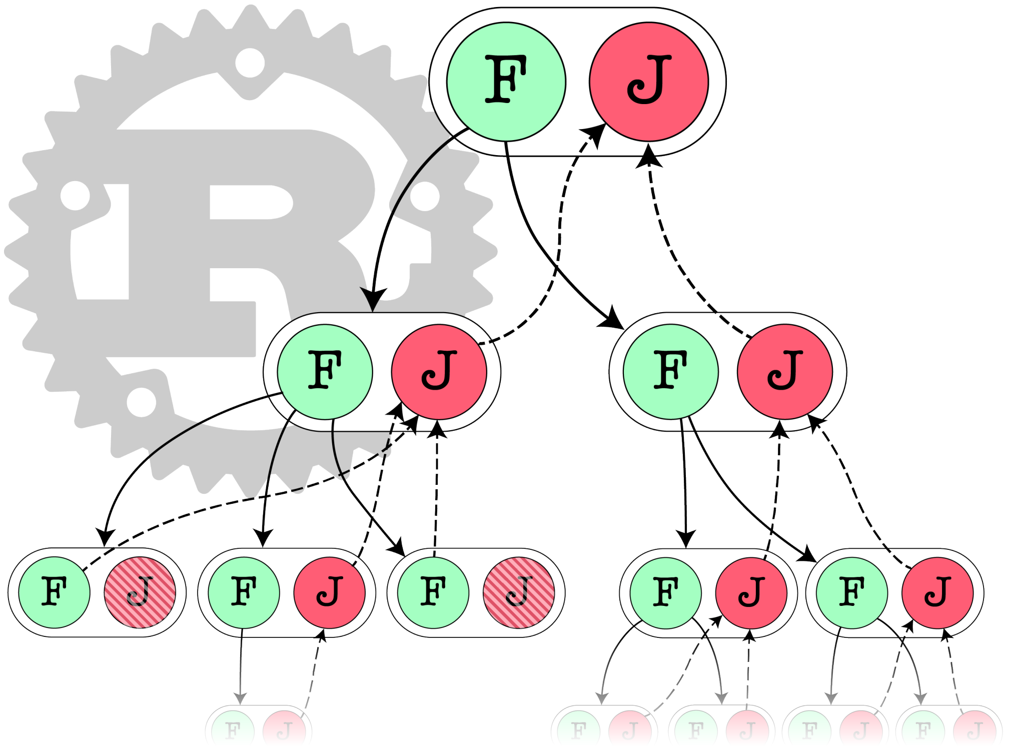
CHALMERS
UNIVERSITY OF TECHNOLOGY



Parallelization in Rust with fork-join and friends

Creating the ForkJoin framework

Master's thesis in Computer Science and Engineering

LINUS FÄRNSTRAND

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

MASTER'S THESIS 2015

Parallelization in Rust with fork-join and friends

Creating the ForkJoin framework

LINUS FÄRNSTRAND

Department of Computer Science and Engineering
Division of Software Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Parallelization in Rust with fork-join and friends
Creating the ForkJoin framework
LINUS FÄRNSTRAND

Academic supervisor:	Josef Svenningsson, Department of Computer Science and Engineering, Chalmers
Industry advisor:	Lars Bergstrom, Mozilla Corporation
Examiner:	Patrik Jansson, Department of Computer Science and Engineering, Chalmers

Master's Thesis 2015
Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A visualization of a fork-join computation that split itself up into multiple small computations that can be executed in parallel. Credit for the Rust logo goes to Mozilla.

Typeset in LaTeX
Printed by TeknologTryck
Gothenburg, Sweden 2015

Parallelization in Rust with fork-join and friends
Creating the ForkJoin framework
LINUS FÄRNSTRAND
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis describes the design, implementation and benchmarking of a work stealing fork-join library, called ForkJoin, for the new language Rust. Rust is a programming language with a novel approach to memory safety and concurrency, and guarantees memory safety through zero-cost abstractions and thorough checks at compile time rather than run time.

Rust is very well suited for computationally intensive applications, but it is lacking some basic tools for creating parallelism, something that is essential for utilizing modern hardware. This paper takes existing algorithms for fork-join parallelism and work stealing and implements them in Rust. The resulting library provides a safe and fast interface to a fork-join thread pool with low work overhead and a good ability to scale up to many processors.

Inspiration for the implementation done during this project comes partly from Cilk, a work stealing fork-join framework for C, but also from other, more modern papers within the same topic.

Three main types of algorithms that fit fork-join parallelism are identified and discussed in this report. This thesis and the design and implementation of ForkJoin, focuses on providing a simple framework for working with these different algorithms, in spite of their internal differences.

Keywords: rust, parallel, parallelization, fork, join, fork-join, work stealing, cilk

Acknowledgements

Firstly, I would like to thank my supervisor at Chalmers, Josef Svenningsson. Svenningsson has been guiding me and the project in the right direction during our weekly meetings, both by questioning my solutions as well as providing experience and knowledge in the field. It was great to bounce ideas with him during our meetings and over emails. He also helped me in my report writing and with more general issues surrounding the master's thesis in general.

I would also like to thank my advisor at Mozilla, Lars Bergstrom, who was the main source of inspiration for the topic of my master's thesis. Bergstrom pointed out what was unsolved in Rust and what the project could have use of. Bergstrom also gave good feedback on my progress and gave me Rust guidance when needed.

The entire Rust community need to be acknowledged. It is a great community with lots of helpful people who instantly help everyone, read their problematic code and come with insightful tips. I want to send out a thank you to everyone in #rust and #rust-internals on irc.mozilla.org who have answered all my questions about borrowed references, lifetime annotations and other topics. *//faern*

I would like to thank my friends, Caroline Hallung and Carl Engström, for reading my report multiple times and providing me with valuable feedback on the content, structure and my usage of the English language.

I would also like to mention my Rust interested friends who made me look into the language and keep my interest for it up during the entire project. Without you I would have no one to share my Rust related findings and tidbits with.

Linus Färnstrand, Göteborg, June 2015

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 The three layer cake . . . 1

1.1.2 The Rust language . . . 2

1.2 Purpose and goals . . . 3

1.3 Scope and limitations . . . 3

2 Theory **5**

2.1 Fork-Join . . . 5

2.2 Work stealing . . . 6

2.2.1 Deque . . . 7

2.3 Cilk . . . 7

2.4 Measuring performance . . . 9

2.5 Memory errors . . . 9

2.6 Rust . . . 10

3 Methodology **13**

3.1 Other implementations . . . 13

3.1.1 Servo's `WorkQueue` . . . 13

3.1.2 Rayon . . . 15

3.2 Deque . . . 16

3.3 API design of ForkJoin . . . 16

3.3.1 Algorithm styles . . . 17

3.3.2 The `ForkPool` class . . . 17

3.3.3 The `Algorithm` struct and `AlgoOnPool` class . . . 18

3.3.4 The `Job` class . . . 19

3.3.5 Tasks . . . 19
3.3.6 Example usage . . . 20
3.4 ForkJoin internals . . . 22
3.4.1 Work stealing . . . 22
3.4.2 Tasks . . . 23
3.4.3 Joining . . . 24
3.5 API changes during the project . . . 24
3.5.1 Lifetimes of task arguments . . . 24
3.5.2 Concrete types . . . 25
3.6 Test and benchmark algorithms . . . 26
3.6.1 Switching to serial execution for small tasks . . . 26
3.6.2 Fibonacci . . . 27
3.6.3 Reduction of trees . . . 27
3.6.4 Nqueens . . . 28
3.6.5 Quicksort and Mergesort . . . 29

4 Benchmarks 31
4.1 Criterion.rs . . . 31
4.2 Benchmark suite . . . 32
4.3 Rust . . . 32
4.4 Hardware for benchmarking . . . 32
4.5 Choosing parameters and arguments . . . 33
4.6 Benchmark results . . . 34
4.6.1 Fibonacci . . . 34
4.6.2 Reduction of trees . . . 35
4.6.3 Nqueens . . . 37
4.6.4 Quicksort . . . 37
4.6.5 Mergesort . . . 38

5 Discussion and conclusions 41
5.1 Benchmark results . . . 42
5.1.1 Fibonacci and reduction of trees . . . 42
5.1.2 Nqueens . . . 43
5.1.3 Sorting . . . 43

6 Future work **45**

Bibliography **47**

A Appendix 1 **I**

A.1 Code used for benchmarks I

A.1.1 Fibonacci I

A.1.2 Reduction of trees II

A.1.3 Nqueens III

A.1.4 Quicksort V

A.1.5 Mergesort VII

A.2 Verbose benchmark results IX

A.2.1 Fibonacci IX

A.2.2 Reduction of trees XI

A.2.3 Nqueens XII

A.2.4 Quicksort XIII

A.2.5 Mergesort XIV

1 Introduction

1.1 Background

Today it is common knowledge within the computer science community that the way to speed up computations is to make them parallel and make use of the many cores that modern processors have. However, writing parallel software is difficult, since it is complex to avoid deadlocks and race conditions, as described by Sutter [1]. Much of what makes it difficult is shared state, where multiple threads of execution access the same memory location. If a thread writes to a memory location at the same time as another thread reads or writes to it, content will not be consistent and likely produce an error. [2]

There are many ways to make programs parallel, some trade control for safety and others only work for very specialized algorithms. No silver bullet for parallelism exists. Different algorithms might be able to make use of different paradigms and styles. Sometimes multiple styles can be used together, but no generic solution exists for parallelizing any algorithm. [3]

1.1.1 The three layer cake

Parallelization comes in many forms, each solving different problems and providing different benefits. However, these different styles of parallel programming can clash with others and create problems if not used correctly together with each other. If used correctly the use of multiple different styles can increase performance. [3]

In the paper by Robison and Johnson, titled *Three Layer Cake for Shared-Memory Programming*, an approach to parallelization is proposed that involves three styles of parallelism arranged in three hierarchical layers. The styles introduced are message passing, fork-join and SIMD. The paper argues that the styles should be used with message passing at the top followed by fork-join, with SIMD at the bottom. The hierarchy means that a layer can use parallelization of layers below itself. The main reason for the hierarchy is that the more restrictions one applies to a computation model, the easier it becomes to reason about how it works at the same time as it becomes less expressive. As such, the least expressive is placed in the bottom where it is easy to reason about it and to use in other, higher, styles without losing correctness guarantees. [3]

1.1.2 The Rust language

Rust (see [4]) is aiming to be a modern, safe and fast systems programming language. Rust is safe from data races between its threads and it is easy to write concurrent message passing programs in it. The drawback is that it only provides kernel threads as a concurrency mechanism. Threads are very low level and are difficult to use directly for parallelizing algorithms. As described in the Theory chapter, general purpose operating system threads introduce too much overhead to be suitable for fork-join parallelism and other styles that handle many small units of computation.

In the library repositories for Rust there are libraries for thread pools and for lightweight threads with much less overhead than kernel threads. These libraries make it possible to spawn lightweight tasks and parallelize programs. However bare threads are, as described, very low level and does not allow algorithms to easily be parallelized.

The topic and goal of this thesis is set partly because of the thoughts and wishes of the project manager of Rust itself, Nicholas Matsakis. Matsakis writes on his blog [5] about Rust's lack of, and need to implement, parallelization primitives. In the blog post he writes about the three layer cake [3], presented earlier and about how he thinks the three layer cake approach will suit Rust perfectly. Matsakis has experimented with some APIs for fork-join[1] and in personal communication[2] he has stated that this is still an unsolved problem. [5]

[1]See *Rayon* under section 3.1.2

[2]Lars Bergstrom, this project's advisor, has been in contact with Matsakis about suitable master's thesis topics within Rust.

1.2 Purpose and goals

The goal of this project is to provide Rust with one of the missing layers in the three layer cake model, and thus to implement tools to simplify parallelizing applications in Rust. The tools to be created include primitives for high performance fork-join parallelism. These tools should allow developers to easily create applications in Rust that are highly parallel, memory safe and run with high performance.

1.3 Scope and limitations

Work stealing is a scheduling strategy where idle worker threads steal work from busy threads to perform load balancing. Many papers have been published on work stealing [6, 7, 8, 3, 9, 10, 11, 12] and its internal data collection, *deque* [13, 12, 8]. This project does not invent new techniques for making work stealing fork-join effective, but instead it explores how to use many of these existing techniques and algorithms to create an effective and fast fork-join library in Rust. It is particularly interesting to implement in Rust since the language has a quite special approach to shared state, see section 2.6 of the theory chapter.

The message passing layer will not be in the scope of this project since it is already implemented. Message passing is one layer in the three layer cake, but Rust contains a high performance implementation of channels that make this layer already available as a way to parallelize Rust applications.

The last layer in the cake, SIMD, will not be reviewed in this thesis. In the library repositories for Rust there are already a couple of SIMD libraries. No work will be directed towards testing these libraries and evaluate if they provide what is needed for the three layer cake model.

2
Theory

Implementing tools for fork-join parallelism in Rust require a good theoretical ground in both Rust and parallelization. There are many papers about fork-join implementations in other languages, and these build the foundations for the tools that this thesis will create for Rust. This chapter will present the research that has been done on parallelism and what is special about Rust and its memory model.

2.1 Fork-Join

Fork-join is a parallelization style where computations are broken down into smaller subcomputations whose results are then joined together to form the solution to the original computation. Splitting of computations is usually done recursively, and the parallelism comes from the property that the subcomputations are independent and solving them can be done in parallel. The computations and subcomputations will be called tasks and subtasks throughout this thesis. [12]

One example of an algorithm that can be implemented in the fork-join style is a recursive algorithm for finding the n:th fibonacci number. Finding the n:th number consists of finding the two previous numbers, $n - 1$ and $n - 2$, and adding these together. Finding the two previous numbers can be done in parallel and represents the fork. Joining is represented by adding the two results together. This algorithm is recursive because finding number $n-1$ consists of forking and finding the $(n-1)-1$ and $(n-1)-2$ numbers etcetera.

Fork-join can be implemented on any system supporting starting subtasks in parallel and waiting for their completion. However, fork-join tasks require very simple scheduling, and they never block except when waiting for subtasks. The simplicity in the tasks make general purpose threads with extensive support for bookkeeping around blocked threads add unnecessary overhead. Tasks in fork-join are usually small and execute for a short period of time, thus the scheduler needs to have a minimal overhead to not slow down the entire computation noticeably. [12]

Many languages have support for fork-join parallelism in one way or another. In C one can for example use the popular Cilk library described in section 2.3 or the more general OpenMP [14] multiprocessing API. In Java it is possible to use the FJTask library by Lea [12], which is implemented with inspiration from Cilk and works in a similar way. When it comes to functional programming, Haskell has a few tools for fork-join parallelism. One tool is the Par monad as described in a paper by Marlow et al. [9], which implements mechanisms for deterministic parallelism in Haskell as well as a complete work stealing scheduler to be used with it.

The different tools for parallelism work a little bit different, and even though they all enable fork-join parallelism they do so in slightly different ways and offer different expressiveness. Cilk only provides a small set of extra keywords on top of ordinary C code. The main keywords in Cilk are *cilk*, *spawn* and *sync* which mark a function as a Cilk procedure, spawn off a parallel computation and wait for all parallel computations to finish (join) respectively. These keywords allow a Cilk procedure to split up control flow in any number of independent threads for parallel execution and then join the control flow again. An implicit join is forced on the return of a Cilk procedure, guaranteeing that the control flow is joined and a call to a Cilk procedure will never leave work or running tasks in the system when returning. Java's FJTask works very similar to Cilk, since it was heavily inspired by it. It allows splitting of control flow and force joining before returning, just like Cilk. [8, 12]

The Par monad by Marlow et al. is more expressive than Cilk and FJTask. The library allows the programmer to express a dynamic data flow graph with multiple listeners for each independent task and can also express producer-consumer parallelism with actors that process messages from each other. In the paper it is demonstrated that it is easy to build algorithmic skeletons that provide functionality specialized for different needs. An example of an algorithmic skeleton from the paper is a divide and conquer skeleton that makes it easy to implement all the algorithms that will be introduced as examples for fork-join algorithms later in this thesis. [9]

OpenMP [14] is another framework for parallelism. It is available for C, C++ and Fortran. The framework consists of a number of compiler directives and library routines that allows the programmer to create both task and data parallelism. OpenMP is much lower level than Cilk and while it allows fork-join parallelism it can also be used for other forms of parallelism. While Cilk provides a very limited set of operations and a clear distinction of what data belongs to what task, OpenMP has many directives for forking control flow in different ways and leaves control over data sharing to the programmer. Managing access to data and what should be accessible where is up to the programmer to define, and is not limited by the framework. [15]

2.2 Work stealing

Work stealing is an algorithm used for scheduling tasks between threads. The algorithm was first introduced by Sleep and Burton back in 1981 [6], and works by letting threads which run out of work steal units of work (tasks) from the other threads. Each thread has its own double-ended queue (deque) where it stores tasks ready for execution. When a thread finishes execution of a task, it places the subtasks spawned by that task, if any, on the bottom[1] of the deque, then it fetches the next task for execution from the bottom of the deque. If a deque becomes empty and the associated thread runs out of work it tries to steal tasks from the top[1] of a randomly selected deque belonging to some other thread. [12]

A property of the recursive divide and conquer algorithms is that they generate large tasks first and split them into smaller tasks later. Thus the oldest tasks, which are located closer to the top of the deque than newer tasks, are likely to be the largest units of work. Because of this a stolen task is likely to generate a lot of work for the thief, reducing the number of steals and thus reducing communication between the workers. Reducing the amount of communication between threads is desired since that includes synchronization, which is expensive. The splitting of tasks is something that continues during the run of the program, not just in the beginning of the computation. A task can be executed at any time, and it is during the run of this task that subtasks are forked. [12]

One of the benefits with work stealing is the trait that worker threads only communicate when they need to. The contrast is work sharing algorithms where the threads that are producing more tasks schedule them on the other threads. Making threads keep all produced tasks for themselves and only interact with other threads when needed improves performance, cache behavior and reduces the need for locking and other synchronization. [16]

2.2.1 Deque

A double-ended queue, a deque, is a data type representing an ordered collection, and is similar to a normal queue. The main difference from a queue is that in a deque, items can be added and removed from both ends of the collection.

In work stealing, a deque is more specialized than the general deque. The bottom of the queue, where both inserting and removing of items are allowed, can only be used by the thread owning the deque. The thread owning the the deque use it as a stack rather than a queue. The top[1] of the queue, where only removal (stealing) of items is allowed, can be accessed by any thread wishing to steal tasks. The goal for a good work stealing deque implementation is to be able to let threads interact with the deque in a thread safe way that adds minimal synchronization overhead. Synchronizing is expensive, and since the deque is so heavily used by the worker threads it needs to introduce as little overhead and waiting for other threads as possible. [13]

2.3 Cilk

Cilk is a runtime system for fork-join parallelism that uses a work stealing load balancing algorithm which is provably efficient. Cilk is implemented as a Cilk-to-C

[1]The front and back of a deque is called its bottom and top in the paper by Chase and Lev [13], and that is the terminology that will be used in this thesis.

source-to-source translator and it produces C as its output which is later linked with the Cilk runtime to produce a program with efficient fork-join capabilities. [8]

There are many existing versions of Cilk. This paper will mostly use and discuss the functionality and implementation of Cilk-5, the latest open source version of the system at the time of the writing of this report.

A multithreaded Cilk program is built up of procedures and threads that together build a computation tree shaped as a directed acyclic graph (DAG). A Cilk procedure is a unit of work that can spawn subtasks, wait for their result and return a value. A thread in Cilk is a part of the procedure, the longest sequence of instructions ending with spawning a subtask, waiting for a subtask or returning from the procedure. [8, 3]

Threads in Cilk are never allowed to block, this follows directly from their definition: they are the longest sequence of instructions that do not block. Since a thread is not allowed to block, a join is not really waiting for the children to complete. Instead a join compiles down to a successor thread that receives the return values of the children and is started when all children have completed. [7]

As long as all processors are busy, the execution path of a Cilk calculation is identical to the serial version of the algorithm. Spawning of a child directly executes the child instead of putting it in the queue. When Cilk reaches a spawn statement it suspends the parent and directly executes the child. While the child is running the parent can be stolen by other processors. If the child is done executing before the parent has been stolen the processor simply continues to execute the parent, allowing an execution path identical to the serial program. [3]

```
cilk int fib (int n) {
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```

Listing 1: Example implementation of fib in Cilk. Finding the n:th fibonacci number in a recursive and parallel way. The code is an exact copy of the demonstration code from the Cilk paper [8].

Listing 1 is a copy of the code used in the Cilk paper do demonstrate how Cilk can be used to find the n:th fibonacci number recursively and in parallel. This algorithm will be used throughout this report to demonstrate how to use different fork-join libraries and frameworks for creating parallelism. The `cilk` keyword is used to mark a function as a Cilk procedure. The `spawn` keyword creates a subtask, or rather makes the current task available for stealing and directly starts executing the subtask. Lastly the `sync` keyword is used for joining and the code after the sync statement will be executed when all subtasks spawned before it have finished executing.

2.4 Measuring performance

The two concepts used in Cilk to measure algorithm execution speed and efficiency is *work* and *critical-path*, but only the work will be introduced here. T_P is defined as the execution time of a computation on P processors. T_1 then is the time it takes to execute the parallel code on one processor and becomes the *work* required by the computation. Further, the execution time of the corresponding serial computation running without Cilk code, but with the same algorithm, is denoted T_S. Using this measurement the *work overhead* is defined as T_1/T_S, and becomes an important measurement of the overhead that will be added to a computation when converted to use the parallelization library. One more term used in the Cilk paper and that will be used here to discuss algorithms and their properties is *parallel slackness*. The term parallel slackness means that the executed algorithm has an average parallelism considerably higher than the number of available processors used to run it. Thus being able to fully use all available processing power. [8]

2.5 Memory errors

When choosing a programming language developers usually have to choose between managing the memory manually or having a built in garbage collector do it for them. Manually managing memory, like in C/C++, provides a certain amount of control. Manually managing memory gives the ability to specify exactly when to allocate and free as well as what to do with it. This freedom comes at the cost of responsibility and error prone code. On the other hand, garbage collectors mitigate the common errors, but at the cost of lost control and usually with some performance penalty. Most modern languages are more or less making a tradeoff between control and safety. [17]

One error that may arise when manually managing memory is *use after free*, also called a *dangling pointer*. Use after free occurs when a reference is taken to some data, and the reference is kept alive even though the data goes out of scope and is freed. After the free the reference points to invalid memory and usage of the reference will yield undefined behavior. [17]

Another common memory problem is data races in parallel code. A data race is when multiple threads reference the same data and one thread writes to that data at the same time as another thread reads or writes to it. During a write operation the data is in an inconsistent state and another read or write of that data will yield undefined behavior. [17]

2.6 Rust

Rust is a new open source programming language spearheaded by Mozilla. The language is a systems programming language aiming to be useable in every place where C and C++ is used today. In contrast to most existing systems programming languages, Rust provides many high level elements such as closures, pattern matching, algebraic datatypes and most notably safe memory management without a garbage collector. [4, 18]

Rust is avoiding the tradeoff between control and safety, described in section 2.5, and instead gets both by taking another approach to memory safety. The goal is to get the performance and control of C/C++ and statically guarantee that the program cannot use the memory in the wrong way. Rust statically checks for memory correctness at compile time and introduces zero runtime overhead while still guaranteeing memory safety. [18, 17]

Rust solves the use after free problem by introducing the concepts of ownership and borrowing. Ownership means that there is always a clear owner of a piece of data, and when the data goes out of scope, meaning the owner is done with it, Rust automatically frees it. Rust can automatically free the data since the compiler has already checked that any references created to it has gone out of scope before the data itself. Borrowing in Rust is the act of creating a reference to a piece of data. The compiler checks that references always go out of scope before the data they are pointing to, giving an error if it does not, thus guaranteeing that references will never be dangling pointers. [18]

Rust solves the issue of data races by only allowing mutation[2] of data when no aliases for that data exist. Either the owner of the data can mutate it or someone holding the exclusive mutable reference to it. A mutable reference to data can be created when no other references exist, and during the lifetime of that reference no other references can be created to that same data. [17]

Ownership and borrowing, how ownership is transferred between functions, is illustrated in listing 2. In the listing, on row 2, a vector is created and it is owned by the variable `vec`. Since `vec` owns the data it can mutate it, shown by a push at line 3. On lines 6 and 7 two references to the data are created with the `&`-operator and one is sent to the function `borrow`. The function `borrow` has now borrowed the data in `vec`, but `vec` is still the owner. When `borrow` returns, the reference goes out of scope, the same happens for `vecref` one row later. When all references have gone out of scope the data is no longer borrowed by anyone. The two references are a demonstration of the possibility to simultaneously create more than one reference to the same data, as long as no reference is mutable.

On line 10 of listing 2 a new mutable reference is created with the `&mut`-operator, and the data in `vec` is lended again. While data is borrowed mutably no other references to that data can exist, in contrast to non mutable references, where any

[2]Changing the data by writing to it.

number of concurrent borrowing can occur. When `borrow_mut` returns the reference goes of scope and `vec` is not borrowed by anyone.

When the function `foo` of listing 2 reaches line 11 and the call to `take` occurs, ownership is transferred to the new function and the variable `my_data` is now the owner of that data. Ownership is transferred because `take` is defined to take its argument by value rather than by reference. If a type taken by value has been marked as copyable, a copy of the argument is created and sent to the function, yielding two instances of the data with two separate owners. Without being marked as copyable the existing object is moved and ownership is transferred. When trying to use `vec` after the call to `take` the compiler will give an error saying that `vec` has been moved and can no longer be used, this is because `take` took over the data and freed it when it returned. Thus `vec` would have been a dangling pointer if used after `take`. [17, 19]

```rust
1  fn foo() {
2      let mut vec = Vec::new();
3      vec.push(13);
4  
5      { // Two references are created and ‘vec‘ is lended to ‘borrow‘
6          let vecref = &vec;
7          borrow(&vec); // Ref goes out of scope when borrow returns
8      } // vecref goes out of scope here
9  
10     borrow_mut(&mut vec);// Lend ‘vec‘ mutably to ‘borrow_mut‘
11     take(vec);           // Ownership transferred to ‘take‘
12     vec.push(37);        // error: use of moved value: ‘vec‘
13 }
14 fn take(mut my_data: Vec<i32>) {
15     my_data.push(99); // ‘my_data‘ is owner, can perform mutation
16 } // ‘my_data‘ goes out of scope and will be freed here
17 
18 fn borrow(vec: &Vec<i32>) {
19         vec.push(10); // error: cannot borrow immutable borrowed ...
20         let element = vec.get(0); // Read is possible.
21 } // Borrowing ends, but ‘vec‘ continues to live in ‘foo‘
22 
23 fn borrow_mut(vec: &mut Vec<i32>) {
24     vec.push(0); // Mutable borrow of ‘vec‘, can mutate.
25 }
```

Listing 2: Examples of ownership and borrowing in Rust

3

Methodology

The library for fork-join parallelism created during this project has been named ForkJoin [20]. This chapter will describe how it was developed and designed.

3.1 Other implementations

Inspiration for the API and the implementation of ForkJoin has been taken from both Cilk and other Rust libraries solving similar or other problems. The following sections describe these Rust libraries, what problems they solve and how.

3.1.1 Servo's `WorkQueue`

The web browser engine Servo [21] is built in Rust and aims to deliver performance far above today's browser engines. Servo claims to rethink the browser engine design from the ground up to optimize for power efficiency and maximize parallelism. One component of Servo is a built-in work stealing fork-join component consisting of a deque implementation and a set of classes for running and managing worker threads. The class `WorkQueue` is the component providing the API to the user.

```
struct WorkUnit<QueueData, WorkData> {
    pub fun: extern "Rust" fn(
            WorkData,
            &mut WorkerProxy<QueueData, WorkData>),
    pub data: WorkData,
}

// push looks the same for WorkQueue and WorkerProxy
fn push(&mut self, work_unit: WorkUnit<QueueData, WorkData>)
```

Listing 3: API signature of Servo's WorkQueue

The public API of `WorkQueue` is shown in listing 3. The only method to add tasks (`WorkUnits`) for computation is through the `push` method. Both the top level submitter of the original computation and the subtasks inside the computation use this `push` method. The task functions themselves, represented by `WorkUnit.fun`, return nothing and there is no way for a task to get the results of submitted subtasks.

Listing 4 contains code for calculating the 40th Fibonacci number using the `WorkQueue`. The algorithm increments a global variable by one every time the algorithm hits the base case, as seen on line 19. After the calculation terminates the global counter is read, yielding the result.

```
 1  fn main() {
 2      let n = 40;
 3      let threads = 4;
 4      let shared_variable = AtomicUsize::new(0);
 5      let mut wq: WorkQueue<AtomicUsize, usize> =
 6          WorkQueue::new("name", threads, shared_variable);
 7
 8      wq.push(WorkUnit{fun: fib, data: n});
 9      wq.run();
10      let result = wq.data.load(Ordering::SeqCst);
11      wq.shutdown();
12  }
13  fn fib(n: usize, p: &mut WorkerProxy<AtomicUsize, usize>) {
14      if n >= 2 {
15          p.push(WorkUnit{fun: fib, data: n-1});
16          p.push(WorkUnit{fun: fib, data: n-2});
17      } else if n > 0 {
18          p.user_data().fetch_add(1, Ordering::SeqCst);
19      }
20  }
```

Listing 4: Finding the n:th fibonacci number using Servo's WorkQueue with 4 kernel threads.

Servo's component does not allow tasks to return any value to their parent task. Everything a task can do is to modify a shared variable or submit subtasks to be executed in parallel. Thus the component does not really allow any join to be performed, rather forking only. `WorkQueue` does not work well for *reduce style*[1] fork-join algorithms. As one might expect the implementation in listing 4 is extremely slow. In practice what happens is that an extremely large number of lightweight tasks try to increment one single shared value. Thus, the `WorkQueue` API might be more useful for algorithms in the *in-place mutation style*[1] or algorithms where the side effects of the execution is the goal.

[1]An explanation of this under section 3.3.1

3.1.2 Rayon

As stated in section 1.1.2, one of the core developers of Rust, Nicholas Matsakis, has looked into parallelization in Rust. At this time he developed a proof-of-concept library doing fork-join parallelism called Rayon. It does not implement work stealing or any kind of load balancing, it simply spawns one kernel thread per submitted task. It mostly shows how he wanted the API to look.

Listing 5 shows the public API of Rayon. A single function, execute, that takes an array of closures with mutable environments. Rayon spawns one thread for each closure, wait for all threads to terminate and then return control to the user.

```
pub type TaskBody<'s> = &'s mut (FnMut() + Send);
pub fn execute<'s>(closures: &'s mut [TaskBody])
```

Listing 5: API signature of Rayon

The API of Rayon makes it difficult to convert to a thread pool implementing work stealing and lightweight tasks on top of a pool of kernel threads. Listing 6 shows what a very naive and basic implementation of finding the n:th fibonacci number would look like. Here the kernel thread running `fib` would have to block while waiting for the call to `execute` to finish, since the mutable closures refer to variables on the stack (`r1, r2`). Thus, it looks difficult to move away from kernel threads to lightweight tasks and keep a similar public API and usage pattern.

```
fn fib(n: usize) -> usize {
    if n < 2 {
        n
    } else {
        let mut r1: usize = 0;
        let mut r2: usize = 0;
        execute(&mut [
            &mut || r1 = fib(n-1),
            &mut || r2 = fib(n-2),
        ]);
        r1 + r2
    }
}
```

Listing 6: Sample implementation to find the n:th fibonacci number in a recursive fashion using the Rayon library.

Lessons learned from looking at Rayon include blocking of threads and the splitting of code before and after the forking. Rayon lets the user execute subtasks inside the task itself and use the results in the same function. A design like Rayon's forces the library to save the stack in some way that allows the parent to continue execution after all the subtasks have completed. A simpler solution is to let the library user break their function into two and let the first one return the forking and the second be called with the forking results.

3.2 Deque

An efficient and fast implementation of the double-ended queue is important for making the work stealing fast. As described in section 2.2.1, an implementation with little synchronization overhead can be vital for the performance of the work stealing algorithm. Such an implementation is demonstrated in the paper by Chase and Lev [13].

The deque implementation used in ForkJoin works as described by Chase and Lev [13], and is available in the Rust package repositories. The implementation is not created as part of this thesis or by the author of this thesis, but the code has been maintained during this project to keep it working as the Rust language and compiler changed prior to the stable release of the language.

3.3 API design of ForkJoin

The main functionality of the ForkJoin library is to manage a thread pool onto which tasks can be submitted and computed results can be fetched. Implementation of the ForkJoin library was done iteratively and the API together with it. The interface was adapted during the entire project to facilitate and enable the implementation of new test and benchmark algorithms.

The API was designed by looking at what functionality other fork-join libraries provided and what different fork-join algorithms needed. Three main types of algorithms that suit fork-join were identified. In this paper the different styles are called *reduce style*, *in-place mutation style* and *search style*. The API and internal design was formed after the requirements of these styles.

Other than the algorithm styles used as a reference, the design was based on the expressiveness and design of Cilk. More specifically the way tasks can fork out in an arbitrary number of independent tasks that later have to be joined together before the task itself returns. A design like this makes it possible to model every computation as a DAG and the execution becomes fairly easy to reason about and get an overview of. Unlike Cilk, ForkJoin has a limitation in the forking, making it possible to only fork and join a task once. In Cilk it is possible to fork, join and then fork again within the same Cilk procedure, something that is not possible in ForkJoin.

There were multiple reasons for selecting Cilk as the reference implementation and source of inspiration for ForkJoin. One reason was that the paper about the three layer cake, which was a large part of the background and motivation behind this thesis, talk about Cilk when discussing the fork-join layer of the cake. Another reason was that Cilk provides a relatively simple and limited way to express fork-join parallelism that still provides the functionality required by most algorithms suitable for this style.

3.3.1 Algorithm styles

Reduce style is where the algorithm receives an argument, recursively computes a value from this argument and returns exactly one answer. Examples of reduce style include recursively finding the n:th Fibonacci number and traversing a tree structure by summing up values in each node. Characteristics of the style is that the algorithm does not need to mutate its argument and the final value is only available after all subtasks have been executed.

In-place mutation style algorithms receive a mutable argument, recursively modifies this value and the result is the argument itself. Sorting algorithms that sort their input arrays are cases of in-place mutation style. Characteristics of the style is that they mutate their input argument instead of producing any output.

Search style algorithms return results continuously and can sometimes start without any argument, or start with some initial state. The algorithm produces one or multiple output values during the execution, possibly aborting anywhere in the middle. Algorithms where leafs in the problem tree represent a complete solution to the problem[2], for example nqueens and sudoku solvers, are search style algorithms. Characteristics of the style is that they can produce multiple results and can abort[3] before all tasks in the tree have been computed.

3.3.2 The `ForkPool` class

The ForkJoin library exposes its API through the class `ForkPool`. A `ForkPool` is a class that manages the threads which are performing the actual work, and is created with the number of worker threads to use. Listing 7 shows `ForkPool`s interface, with its two methods, and lines 5-6 of listing 8 show how to create a `ForkPool` that is ready to accept work.

Computations are submitted to the `ForkPool` by creating an `AlgoOnPool` via the `ForkPool::init_algorithm` method, and calling `schedule` on that object. Thus a `ForkPool` is never used directly, except for creating `AlgoOnPool` instances which are then used to schedule the actual work.

[2]Unless the leaf represents a dead end that is not a solution and does not spawn any subtasks.
[3]The current version of ForkJoin does not support aborting.

```rust
type TaskFun<Arg, Ret> = extern "Rust" fn(Arg)
    -> TaskResult<Arg, Ret>;
type TaskJoin<Ret> = extern "Rust" fn(&[Ret]) -> Ret;
type TaskJoinArg<Ret> = extern "Rust" fn(&Ret, &[Ret]) -> Ret;

enum TaskResult<Arg, Ret> {
    Done(Ret),
    Fork(Vec<Arg>, Option<Ret>),
}
struct Algorithm<Arg: Send, Ret: Send + Sync> {
    pub fun: TaskFun<Arg, Ret>,
    pub style: AlgoStyle<Ret>,
}
enum AlgoStyle<Ret> { Reduce(ReduceStyle<Ret>), Search }
enum ReduceStyle<Ret> { NoArg(TaskJoin<Ret>), Arg(TaskJoinArg<Ret>) }

impl<Ret> Job<Ret> {
    pub fn try_recv(&self) -> Result<Ret, ResultError> {...}
    pub fn recv(&self) -> Result<Ret, ResultError> {...}
}
impl<...> ForkPool<'a, Arg, Ret> {
    pub fn with_threads(nthreads: usize)
        -> ForkPool<'a, Arg, Ret> {...}

    pub fn init_algorithm(&self, algorithm: Algorithm<Arg, Ret>)
        -> AlgoOnPool<Arg, Ret> {...}
}
impl<...> AlgoOnPool<...> {
    pub fn schedule(&self, arg: Arg) -> Job<Ret> {...}
}
```

Listing 7: The signatures of the public methods for creating a `ForkPool`, instantiate an algorithm on it and `schedule`, where work is submitted to the ForkJoin library, together with the main types of the public API.

3.3.3 The `Algorithm` struct and `AlgoOnPool` class

A fork-join algorithm is represented in the library by an instance of the `Algorithm` structure. The structure's definition is shown in listing 7, and includes a task function pointer and an algorithm style enum. As described in section 3.3.1, the style is one of reduce, search or in-place mutation. In ForkJoin, in-place mutation is represented by the reduce enum, as these are treated in the same way by the library. A reduce style algorithm also specifies the type of, and a pointer to, the join function.

The `AlgoOnPool` class is the representation of a specific algorithm on a specific `ForkPool`. The class is a thin wrapper around the `ForkPool` and contains an `Algorithm` instance, which is the information needed to convert an argument to a `Task` that can be executed on the pool. The class only has one method, `AlgoOnPool::schedule`, that takes an argument of the type specified as `Arg` on the algorithm, and returns immediately with a `Job`.

3.3.4 The `Job` class

A `Job` is a handle to a computation, through which the result[4] can be read. The class has two methods, shown in listing 7, that allow results to be fetched. The two methods are similar and the only difference is that `recv` blocks until a result arrives and `try_recv` returns immediately no matter if there is a result available or not.

The destructor of `Job` blocks and waits for the background computation to complete before proceeding. Thus, the owner of a `Job` object will be blocked if it lets the handle go out of scope before the computation has finished. The `Job` cannot take the shortcut of killing the worker threads to achieve its goals, the pool needs to be kept alive since other jobs might occur simultaneously or be scheduled in the future. The current design of the `Job` and its requirement to live until the computation completes is a step towards a design that was intended, but not completed. Section 3.5.1 discusses how the type system of Rust forces references in task arguments to live longer than the entire `ForkPool`. However, the goal is to create a design where the `Job` bears the responsibility of the lifetime and the argument's lifetime only needs to outlive the `Job`, not the entire pool. Also, if an abort mechanism is implemented in the future it will allow the destructor of `Job` to cancel the computation and can thus return almost instantly without leaving tasks in the pool that keeps it busy.

3.3.5 Tasks

At a high level the fork-join problem can be viewed as a tree with every node represented in ForkJoin by a `Task`. If the system is viewed on a lower level the problem is instead a directed acyclic graph (DAG) with every `Task` representing two nodes. On the DAG level a task first represents the forking node splitting the problem into multiple subproblems (subtasks). Secondly the task represents the join node that collects the results of the subtasks and propagates the results toward the top join node represented by the root task of the computation. A possible execution path of the problem tree is illustrated in figure 3.1. It is important to note that figure 3.1 is not cyclic, the task and join functions and just drawn together in pairs to show that they are related, not connected. The figure illustrates the computation as both the tree level and the DAG level.

Tasks can either compute to a base case in the algorithm or fork into multiple subtasks. A task indicates said choice by returning one of two alternates from the

[4]Possibly multiple in the search style algorithms that can return multiple results anytime during the execution.

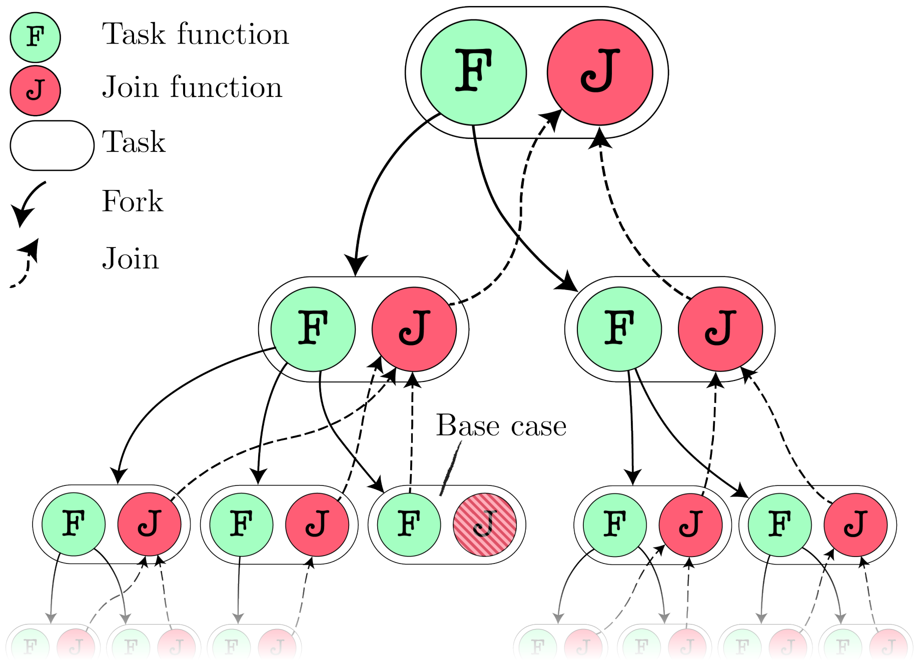


Figure 3.1: An illustration of a computation tree forking and later joining results together. This is an illustration of a reduce style or in-place mutation style algorithm.

enum `TaskResult` from their task function. The values are `Done(Ret)` and `Fork(Vec<Arg>, Option<Ret>)` respectively.

The design of the task shows the user that the functions are not allowed to block and how to split up the code. A first function returns either a computed value or an instruction for forking. Since the task functions provide no way of forking or propagating values up before returning, users will have to return to continue working through the DAG computation.

3.3.6 Example usage

A demonstration of how an algorithm can make use of fork-join parallelism with the help of the ForkJoin library is shown in listing 8, where a recursive implementation of finding the n:th fibonacci number is implemented.

The base of the algorithm is its task and join functions. They are named `fib_task` and `fib_join` in the code. The join function simply sums up the values of all the subtasks it was spawned together with. The task function either returns `Done(value)` on the base case or `Fork(arguments_to_subtasks)`. In fib, the base case is when the remaining work is so small that it is more efficient to run a serial implementation than to fork. Forking is done in all cases that is not the base case, and results in two more tasks that can be executed in parallel.

```rust
pub fn fib(n: usize, threads: usize) -> usize {
    let forkpool = ForkPool::with_threads(threads);
    let fibpool = forkpool.init_algorithm(FIB);
    let job = fibpool.schedule(n);
    job.recv().unwrap()
}

const FIB: Algorithm<usize, usize> = Algorithm {
    fun: fib_task,
    style: AlgoStyle::Reduce(ReduceStyle::NoArg(fib_join)),
};
const SERIAL_THRESHOLD: usize = 20;

fn fib_task(n: usize) -> TaskResult<usize, usize> {
    if n <= SERIAL_THRESHOLD {
        TaskResult::Done(sequential_fib(n)) // Small task
    } else {
        TaskResult::Fork(vec![n-1,n-2], None)
    }
}
fn fib_join(values: &[usize]) -> usize {
    values.iter().fold(0, |acc, &v| acc + v)
}
```

Listing 8: Sample code for creating a work stealing thread pool in the ForkJoin library, submitting a job to it and waiting for the result. The code shows how a task is divided into the task function that can return or fork and the join function that joins together results from a fork.

The function `fib` in listing 8 shows how the ForkJoin library is used with an algorithm implementation. In detail, it shows how a thread pool is created via the `ForkPool` class, and how a computation of the n:th fibonacci number is submitted to the pool and the result returned when available. The first line of the function will create a `ForkPool`, that will start the `thread` number of worker threads in the background. The next line creates the `AlgoOnPool` instance that allows fib computations to be carried out on that specific pool. The third row schedules a computation on the pool, this is where the threads start working in the background and when `fib_task` and `fib_join` will start getting called by the pool. On the last line of the function, the `Job::recv` function is called and execution will block until the computation is done and a result is returned.

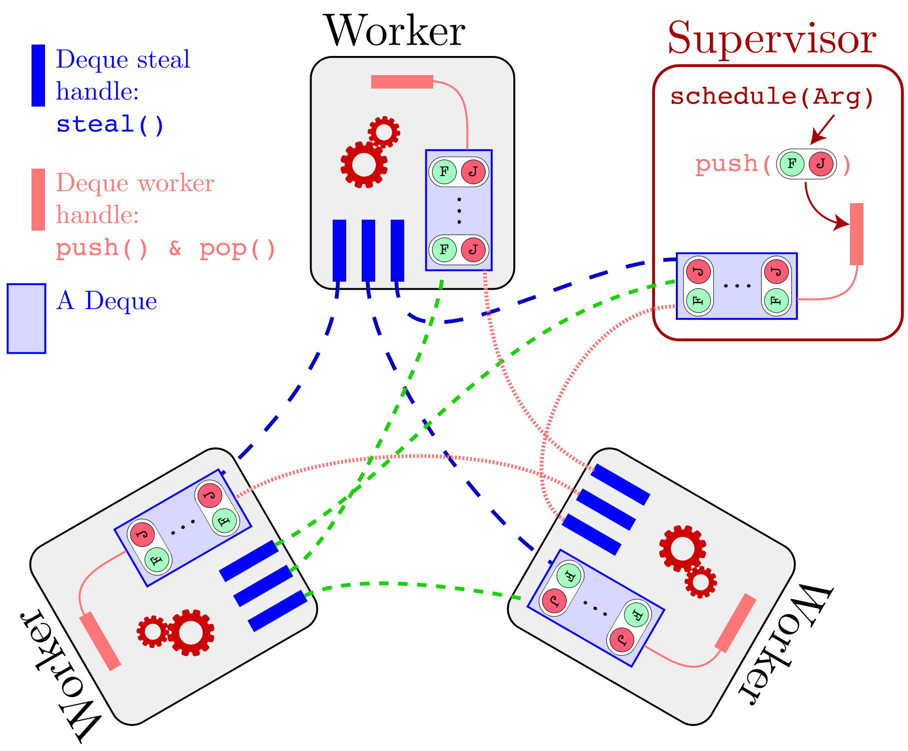


Figure 3.2: An illustration of worker threads, their supervisor and the different handles they have to the deques. The supervisor has a deque unto which it pushes newly scheduled tasks. Every worker has a steal handle to every other workers deque and one steal handle to the supervisors deque. Mostly the workers use their private worker handle to their own deque to `pop` tasks to execute and `push` tasks that are forked. However, when a deque becomes empty and the worker have no tasks, it tries to steal from a randomly selected steal handle.

3.4 ForkJoin internals

The top handle to the thread pool is the `ForkPool`. In the background the `ForkPool` creates a `PoolSupervisor` that, in turn, spawns the given amount of worker threads.

When work is scheduled on the `ForkPool` it sends the task to the `PoolSupervisor`. The supervisor puts the task on its own deque, from here it will automatically be stolen and executed by an idle worker thread. If the pool is idling, meaning all worker threads are waiting for a message from the supervisor, then the supervisor wakes all of them up, so they start stealing and execute tasks.

3.4.1 Work stealing

Load balancing in the thread pool works by letting worker threads without any tasks to execute steal tasks from busy worker threads. As soon as a worker thread runs out of tasks to execute in the local deque it goes into stealing mode. An illustration of the relationship between workers can be seen in figure 3.2. Stealing will continue until a task is successfully stolen or until all worker threads are in stealing mode, thus guaranteeing there are no tasks left in the entire pool.

Stealing starts out at a random victim and then tries all the other victims. Each worker thread has a steal handle to the deques in all the other worker theads in the same pool and one to a deque owned by the pool supervisor, as illustrated in figure 3.2. All steal handles are located in a vector in the worker thread's structure. First a random index in the vector is selected, stealing starts at this index. If stealing fails it continues with the next steal handle in the vector, and iterates like this until all handles have been tried once or a task was successfully stolen. One iteration of the steal handles is called one *steal round*, a term used to describe the backoff algorithm.

In stealing mode a backoff algorithm is active to prevent the worker thread from creating heavy load while stealing. Between each consecutive steal round a counter is incremented, and if the counter is above a hardcoded threshold the worker thread goes to sleep for a short period of time before starting the next round of stealing. Sleeping start off at 10 μs and is increased by 10 μs for every steal round. Both the iteration counter and the sleep time is reset every time the worker thread enters stealing mode.

The work stealing algorithm is also responsible for detecting when the thread pool runs out of work, and stop all the worker threads when this happens. Before a thread goes to sleep in the backoff algorithm it increments an atomic counter shared by all the worker threads, and when the thread wakes up it checks the value of the atomic counter. If the value is the same as the number of worker threads in the pool it means all worker threads are currently sleeping, thus no worker threads have any tasks. The thread then cancels the stealing and becomes idle, waiting for a message from the pool supervisor. If the counter is less than the number of worker threads it decrements the counter, thus removing itself, and then continues with the stealing process.

3.4.2 Tasks

Tasks consist of an `Algorithm` instance, an argument and an object describing what to do with the result of the computation, called the join. Executing a task includes running the task function, located in the algorithm, with the supplied task argument, then either propagating the result up the computation tree or forking down. If the task function returns a `TaskResult::Done(Ret)` the execution has reached a leaf in the computation tree and the result is handled according to the task's join. If the task function returns a `TaskResult::Fork(Vec<Arg>, Option<Ret>)` the worker thread creates one subtask for every argument in the vector and pushes them onto the local execution deque. Code illustrating forking for `fib` can be found in listing 8, where the task forks on the arguments `n-1` and `n-2`.

3.4.3 Joining

Joining computed results works differently depending on the algorithm style. Search style algorithms send their results directly to the `Job` handle through a channel in the task's join-field. Reduce style algorithms have a more complex mechanism called the `JoinBarrier`.

When tasks in reduce style algorithms fork into n subtasks a `JoinBarrier` is created for all the subtasks to deliver their results to. The `JoinBarrier` have a preallocated vector of size n to store return values in. The barrier also has an atomic counter initialized with the value n and indicates the number of missing values. Each subtask is given a reference to this shared `JoinBarrier` and a pointer to a unique position in the barriers result vector. Upon subtask completion the resulting value is directly inserted into the memory position pointed to by the unique pointer, and it then decrements the atomic counter in the barrier. When a task detects that the counter reach zero it knows it is the last to complete and thus it executes the task joining function given to the barrier from the parent task creating it. The join function is being executed with the return value vector as argument. The implementation of the JoinBarrier takes inspiration from how Cilk [7] collects values from subtask closures.

Optionally one extra value can be passed directly from the task function to the join function. The extra value is passed via the second argument in `TaskResult::Fork(Vec<Arg>, Option<Ret>)`. If the value is present and the algorithm specifies `ReduceStyle::Arg` it will be stored in the `JoinBarrier` while the subtasks are executed and later sent to the join function together with the results of the subtasks.

3.5 API changes during the project

Just after studying other frameworks and solutions for fork-join parallelism there was a plan and intended design for the ForkJoin library. Some of the ideas could not be implemented within the timeframe of this project, for reasons described below.

3.5.1 Lifetimes of task arguments

There is an inherent problem with the planned design of the ForkJoin library and the way lifetimes work in Rust. In Rust, if an object containing references is created, the references must outlive the object containing them, so that the object cannot possibly reference invalid memory. In a larger perspective the lifetime restriction makes it impossible to submit tasks to a `ForkPool` when the tasks contain references living shorter than the entire thread pool, because then the compiler cannot guarantee that the pool does not try to dereference a dangling pointer. Originally the plan was that a thread pool was set up when the program started and lived until the program terminated, so that kernel threads would only have to be spawned once. Rust does not allow said design if the library should be allowed to receive references with lifetimes shorter than the entire program without using unsafe code that works around the restrictions.

What the lifetime restriction means in practice is that algorithms taking references in their task arguments rather than ownership of the argument must create a new `ForkPool` with a shorter lifetime than the referenced data. The problem occurs with sorting algorithms, that take a mutable reference to the data rather than the ownership of the vector.

Most other libraries with similar problems have limited their input to objects without references living shorter than the the entire programs execution. A decision was taken to allow ForkJoin to take references with arbitrary lifetimes despite the problem described above, this because the goal to one day make it work around the issue and provide the possibility to use references freely in the tasks.

The problem can be viewed as three requirements that are impossible to satisfy at the same time:

1. Use Rust without unsafe code and stick to what the original type system allows.
2. Allow data containing references with arbitrary lifetimes to be sent in and out of parallel computations.
3. Use the same thread pool instance for computing all tasks, from the start of the program until its termination.

As described above, these three requirements cannot be fulfilled at the same time by the ForkJoin library. The current version of the library does not satisfy the third requirement, since the compiler will not allow a thread pool living longer than the references sent to it. Most other parallel libraries in Rust do not satisfy the second requirement, forcing only owned data to be sent around. The goal for ForkJoin is to satisfy the second and third requirement but not the first. The ForkJoin library already uses unsafe code in many places, for example to allow mutable pointers directly into the `JoinBarrier`'s result vector. Using unsafe code is not something that should be avoided in Rust, but the amount of code that is unsafe should be kept relatively low for easier debugging.

The lifetime restriction described is an interesting finding in this thesis and it shows a case where Rust's memory model stops the developer from doing what could have turned into a memory error in other languages. Rust forces the developers to find a safer solution to problematic code or mark it as unsafe and thus highlight the code as a potential source of problems during future debugging.

3.5.2 Concrete types

As stated above, the initial design goal was to be able to create a `ForkPool` at the start of the program and keep that same instance until program termination, letting all jobs that can benefit from fork-join use the same thread pool. Apart from the problem of lifetimes and references in arguments described previously the design goal was also not possible without unsafe code because of the strict type system. A `Task` has generics to specify what type of argument it takes and what type the return value has. The signature has this form: `Task<Arg, Ret>`. These types have to be specified to a concrete type when using the `Task`. A `WorkerThread` contains a deque with all its `Task`s to be executed. Thus the `WorkerThread` needs to specify input and output types of the tasks in its deque at creation. Part of the declaration of the `WorkerThread` struct can be seen in listing 9.

The type restriction described above limits one instance of a `WorkerThread` to one input type and one output type, thus making it impossible to use the same `WorkerThread` for two algorithms that need different input or output types.

```
pub struct WorkerThread<Arg, Ret> {
    // ...
    deque: Worker<Task<Arg, Ret>>,
    // ...
}
let x: WorkerThread<&Tree, usize> = /*...*/;
```

Listing 9: Parts of the definition of the `WorkerThread` struct to show how types have to be concretely given on instantiation.

3.6 Test and benchmark algorithms

At least one algorithm in each algorithm style was implemented to test the ForkJoin library's performance and correctness as well as to demonstrate its expressivness and ease of use. The algorithms were chosen after how well they represent each style, how well known their algorithms are and on how much performance comparison data was available from other fork-join libraries. The code for each algorithm can be found in appendix A.1 of this thesis.

3.6.1 Switching to serial execution for small tasks

As described in section 2.3 of the theory chapter, it is important that a computation generates a sufficiently large number of independent tasks, to achieve parallel slackness. Parallel slackness requires that there are enough tasks to keep all worker threads busy in order to fully utilize the available processing power. On the other hand, when every worker thread has enough work, forking new tasks does not further increase parallelism but rather creates more overhead. Therefore, to maximize performance a fork-join algorithm should divide into enough tasks to create work for all processors, but not much more.

A common technique to balance the parallel slackness, and get good performance, is to use a *granularity threshold* [12]. Implementing a granularity threshold means that the algorithm, at some point, switches over to serial execution rather than forking to more tasks. The technique only works for algorithms where a task can determine approximately how much work there is left to do and decide if it is more efficient to switch over to a serial version of the algorithm or not.

3.6.2 Fibonacci

Calculating the n:th Fibonacci number recursively falls into the reduce style category. The algorithm includes close to zero work, almost only forking and joining overhead. Therefore it is a good example to show the amount of overhead the tested library introduces compared to a normal recursive function call or compared to other parallelization libraries.

The fibonacci algorithm must have a granularity threshold where it runs serial code instead of forks. Without the threshold the algorithm would generate a very large amount of tasks containing almost no work to be performed, thus giving unnecessarily high parallel slackness at the cost of performance.

3.6.3 Reduction of trees

Running a reduce style algorithm on a tree structure of unknown shape, like the one possible to create with the data structure in listing 10, poses a special challenge. The algorithm used is very simple, iterating over a tree structure summing up the values in each node. The problem is with granularity and switching to serial code. In the fibonacci algorithm a task can decide to run a serial implementation of `fib` when the argument is smaller than some threshold or a certain depth of the problem tree has been reached. Thus it avoids creating many small tasks that add more to overhead than they gain in parallelism. However, when the algorithm knows nothing about the shape of the tree it cannot make a good judgement about whether or not to switch over to a serial algorithm or to fork into subtasks for an arbitrary node. See figure 3.3 for an example of an unbalanced tree. In this algorithm everything that is known when inside a given node is the number of direct children that node has and possibly how deep down the tree the node is positioned, it knows nothing about the remaining depth. Thus switching to serial

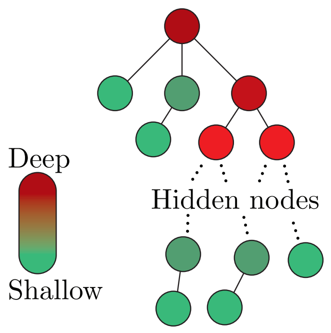


Figure 3.3: A tree where some children are deep trees and some not. A good example showing it is impossible to select a threshold, without examining the tree, that guarantee good execution.

code might turn out to greatly reduce the parallelism if the node where the serial code is started contained a majority of the entire tree.

```
pub struct Tree {
    value: usize,
    children: Vec<Tree>,
}
```

Listing 10: Definition of tree structure used in tree reduce algorithm.

3.6.4 Nqueens

The problem called nqueens can be implemented both as a reduce style algorithm or a search style algorithm. In this thesis it is implemented as a search style algorithm to provide an example of that style. The problem consists of placing n chess queens on a chessboard of size n x n without any queen able to attack any other queen. Implementing nqueens in the search style means sending a complete placement of the n queens to the `Job` responsible for that computation as soon as it is found. Implementing it in the reduce style would mean returning a vector of all valid placements after all possible placements have been evaluated.

Cilk's [7] performance benchmarks on nqueens is one aspect that makes it an interesting algorithm to test. In the Cilk paper, a close to 1:1 performance ratio between the serial code and the parallel code running on one thread is shown. Reaching a ratio close to 1:1 means introducing little overhead in the parallelization and thus shows an efficient implementation.

The ForkJoin implementation is not fully comparable with the Cilk implementation. In Cilk the abort feature is used and the entire computation is canceled when the first valid placement of queens is found. Since ForkJoin does not support aborting a scheduled job the algorithm implemented in this project uses search style, but benchmarks the time it takes to find all possible placements of queens and compare it with a serial reduce style implementation.

Nqueens is the only algorithm in this thesis, besides sumtree, that demonstrates forking into a variable amount of subtasks, instead of always forking into two subtasks. A given task, that does not return a base case, forks into anything between 0 and n subtasks, where n is the size of the problem being computed. The algorithm also demonstrates a dead end or a leaf of the computation tree not representing a solution in the search style. A leaf in the computation tree of nqueens is either a solution with n queens placed, or a task where x queens are placed and there is no valid placement of queen $x + 1$. In such a dead end leaf the algorithm return the fork value, but with zero fork arguments to indicate that no further work should be done.

3.6.5 Quicksort and Mergesort

Sorting an array of values and terminating with the sorted values located in the same array is a perfect example of an in-place mutation style algorithm. Quicksort and Mergesort are good candidates for illustrating this style of fork-join parallelism since they both use a divide and conquer style approach dividing the values in two parts and sorting the parts individually.

Both quicksort and mergesort have inherent problems that prevent them from scaling linearly as more worker threads are added. Quicksort does a substantial amount of work serially before the forking can start, and Mergesort needs a long serial computation at the end of the algorithm.

Quicksort does not scale well because of the serial code in the start of the computation. The algorithm starts by selecting a pivot element and then partitions all the elements with respect to this element. After the partitioning is done the algorithm forks into two tasks and the computation can continue on two threads. Since the first partition involves reading every value in the input array and even moving some of them, this step greatly reduces the algorithm's ability to scale to many worker threads.

A problem similar to the one in quicksort also exists for mergesort, but occurs in the final step of the computation rather than in the first. Mergesort starts out by recursively splitting the input array in two and forks, thus creating instant parallelism, unlike quicksort. On the other hand, in the join operation of mergesort all the values need to be read and moved around serially, making the last join operation serially read and move every value in the array.

Similar to how fib works, mergesort also needs a granularity threshold to stop it from splitting the array recursively until every part is only one element long, creating many tasks so small that they only add overhead. The implementation used here switches to sorting with the serial version of quicksort for input arrays smaller than the selected threshold. The industry standard is to combine mergesort with insertion sort for small arrays, here quicksort was choosen because it was already implemented and might provide benchmarks that can be more fairly compared with quicksort.

4

Benchmarks

During the development of ForkJoin, algorithms suiting the different styles presented were developed to verify that it was possible to express them in the library. After the library stabilized and at least one algorithm from each style was implemented, and worked as intended, focus was shifted to improving performance. This chapter will describe how the library and the algorithms written in it were benchmarked, as well as present the results and compare them with other similar libraries.

4.1 Criterion.rs

Criterion.rs [22] is a Rust port of the popular Haskell benchmarking library Criterion [23]. Criterion.rs provides tools to easily benchmark Rust code and calculate various statistics about the running time of the code. The data generated include mean and median execution times as well as graphical plots. The advantage of using Criterion.rs over manually timing the execution time of code is that it does a lot of advanced measurements automatically. The library first runs *warm up*, which is executing the code to be benchmarked multiple times before timing starts. Warm up is performed since the first executions are usually slower because of data missing in the caches as well as clocked down CPUs. After warm up, Criterion.rs collects multiple samples of the time it takes to run the code. Using all the samples it detects outliers and performs tests of how stable the measured times are, thus showing how reliable the numbers are.

The original criterion.rs implementation did not provide all the needed functionality, so this project uses a local fork of the project with improvements. The original implementation focused on testing one function with many different inputs, for example, testing sequential fib with a list of arguments and plotting the different execution times. When benchmarking sequential vs parallel execution, the benchmarks are more focused on comparing different functions with identical arguments. Therefore a new method of benchmarking was added, allowing the user to pass a list of functions and a single argument instead of the inverse.

In addition to an extra measurement method, extra statistical output was added to criterion.rs. The library was extended with the functionality to save csv files with percentile data of the sample distributions for easy import and usage in other programs.

4.2 Benchmark suite

To make benchmarking simple and fast, an automated benchmarking suite based on criterion.rs was developed. The suite contained all the algorithms under test and ran them based on the supplied command line arguments. Arguments to the benchmarking suite include criterion.rs sample size, what thread pool sizes to benchmark, a list of algorithms to benchmark and a list of algorithm arguments for each algorithm.

4.3 Rust

Rust, being a young and actively developed language, changes rapidly. The language changes in syntax as well as standard library features and performance of code generated by the compiler. To make the results fair and stable, a specific version of the rust project was chosen to be used for all benchmarks. The version of Rust used here is the prebuilt version from the nightly development channel at 2015-05-14.

```
rustc 1.1.0-nightly (e5394240a 2015-05-14) (built 2015-05-13)
```

The compiler was configured to use maximum optimization, with the flag `-C opt-level=3`, during the compilation of the ForkJoin library and the benchmarking suite.

4.4 Hardware for benchmarking

To test how well the library and the algorithms written in it scale with many worker threads, benchmarks has been executed on an Amazon EC2 instance with 36 cores running Ubuntu GNU/Linux. All benchmarks are performed on the exact same configuration to obtain comparable results.

One concern with the cores in the virtual EC2 instance was that they would be virtual and share underlying physical cores, thus putting load on one core would lower the performance of others. To test this concern, a single core benchmark using the same fibonacci algorithm as used for the rest of the tests were performed during both idle and almost full load. 32 out of the 36 cores were being kept busy by running 32 instances of the command `cat /dev/urandom > /dev/null`. The results of the benchmarks were that on an idle server the median time was 1137 ms, and on the server under load the exact same benchmark took 1200 ms. These benchmarks show an increase of 5.5%, thus the stated concern was not a problem.

4.5 Choosing parameters and arguments

Benchmarks were executed with less worker threads than physical cores to minimize the risk of threads getting swapped out. If the ForkJoin library occupies all processor cores with calculations it is unavoidable that the operating system will swap out at least some of the threads some time during the execution to let other programs run. Having worker threads swapped out increase the wall clock time of the computation and makes the benchmark results unfair. The results are unfair since benchmarks using fewer worker threads than physical cores do not get swapped out and comparing their results would yield a scale up lower than the algorithm actually have.

The sample size was chosen large enough to make the measurement accurate, and small enough to not take too long time. A sample size of 20 was selected for the benchmarks in this report. Violin plots of the sample distributions, validating that the measurements are reasonably stable, can be found in appendix A.2.

Too small computations do not spread to all worker threads before they are completed, therefore benchmarking on many threads require large problems to yield enough parallel slackness[1] and correct scale up results. The parallel slackness need to be high enough to put all threads to work. However, algorithms such as fib easily generate unnecessarily much parallelism if no granularity threshold[2] is implemented. Because of this, empirical measurements have to be performed in order to find algorithm argument sizes and granularity thresholds that give the best results.

Code for execution statistics was added to the worker threads allowing them to print how many tasks they had executed during a computation, how many tasks they had stolen and how much time they had spent sleeping. The statistics allowed testing of different parameters on the algorithms under test. The problem size was chosen to be the smallest that still made all threads perform approximately the same amount of work, and the granularity threshold was selected to be the argument size for which serially computing the result took approximately the same time as performing a fork operation in the ForkJoin library.

[1]See the theory chapter on Cilk (2.3) for the definition.

[2]See section 3.6.1

4.6 Benchmark results

Full benchmark results in tables and plots can be found in appendix A.2. This chapter will present a summary of the results and reason about them.

Algorithm	Size	T_S	T_1	T_8	T_{32}	T_1/T_S	T_1/T_8	T_1/T_{32}	T_S/T_8
fib	42	1060	1080	142	50.7	1.02	7.61	21.3	7.46
fib no threshold	32	8.64	604	91.6	39.3	69.9	6.59	15.4	0.09
sumtree	23	31.9	825	131	58.2	25.9	6.30	14.2	0.24
nqueens	12	1500	1690	231	153	1.13	7.32	11.0	6.49
quicksort	10^7	1470	1490	261	167	1.01	5.71	8.92	5.63
mergesort	10^7	1240	1400	290	297	1.13	4.83	4.71	4.28

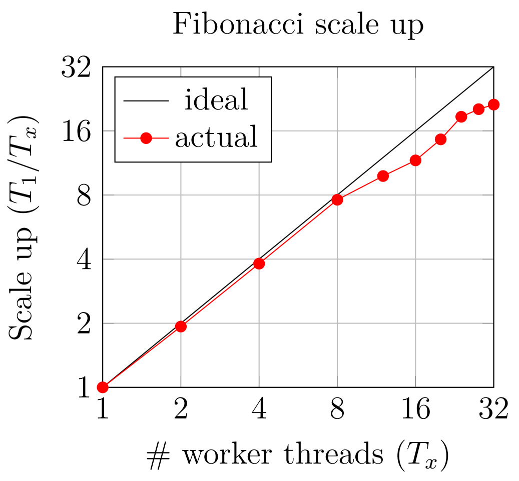
Figure 4.1: Table of benchmark results for all algorithms. All times are the median times of the 20 samples collected. Violin plots of all samples can be found in appendix A.2. All times are in milliseconds. The work overhead described in section 2.3 is presented in the column labeled T_1/T_S and how well the algorithms scale to multiple worker threads can be seen in the columns T_1/T_8 and T_1/T_{32}. How well the algorithm perform on 8 threads compared to the serial code can be seen in the column T_S/T_8.

4.6.1 Fibonacci

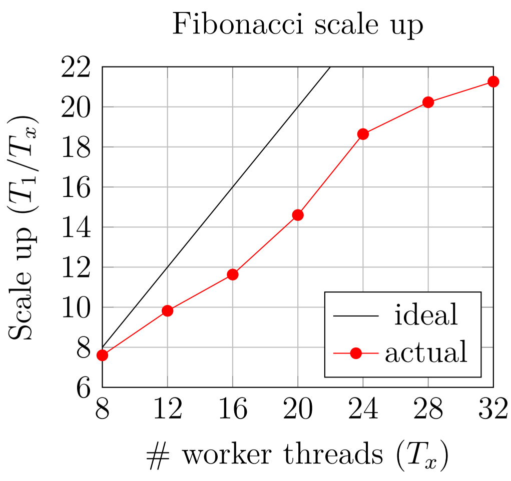
Benchmarks were performed to find what argument to the sequential fibonacci algorithm resulted in a computation time approximately the same as that of a single forking operation in the ForkJoin library. Calculating the 20th fibonacci number gave an almost equal execution time and was chosen as the granularity threshold. The code, including the threshold, can be seen in appendix A.1.1 and detailed performance data in A.2.1.

As can be seen in the results in table 4.1, fib with a reasonably selected threshold have close to zero work overhead. Without the granularity threshold the algorithm generates a large amount of tasks with almost no work in and the overhead becomes apparent.

When it comes to how the algorithm scales to multiple worker threads, the table shows that close to linear scale up is achieved up to 8 worker threads. After 8 threads the scaling flattens out and when the computation is carried out on 32 worker threads the time is 21 times faster than on 1 thread.



(a) Log-log presentation of all measured values.



(b) Linear presentation of the part diverging from the ideal.

Figure 4.2: Graph showing how the computation time of the fibonacci algorithm scale as more worker threads are added.

The results of this implementation can be compared with the ones in Cilk 5. The fibonacci algorithm results presented in the Cilk 5 paper [8] are for an implementation without granularity threshold that forks until the argument is less than 2. However, even without a threshold, Cilk manages to have a work overhead of 3.63 and a scale up factor of 8.0 for T_1/T_8.

4.6.2 Reduction of trees

Short of fib without a threshold, sumtree is the only algorithm with a work overhead over 2. The scaling of the algorithm is similar to fib without threshold, after 8 threads this algorithm also fails to scale much further.

The sumtree algorithm is the only algorithm[3] that does not have better performance in the parallel execution than in the serial execution no matter how many worker threads are added. This clearly indicates that as long as the implementation does not improve in some way there is no case where converting the algorithm to the parallel version would be desired.

As discussed in section 3.6.3, the shape of the tree is important. One reason is that if the writer of the sumtree algorithm does not know anything about the shape of the trees, a threshold that will guarantee good performance cannot be set. Another reason the tree shape is important is because of the load balancing. When iterating over a balanced tree, almost no load balancing needs to be done since all children contain the same amount of work and all threads will be busy approximately the same time. On the other hand, if the tree is heavily unbalanced, some threads that steal small portions of the tree will run out of work fast and have to go back to

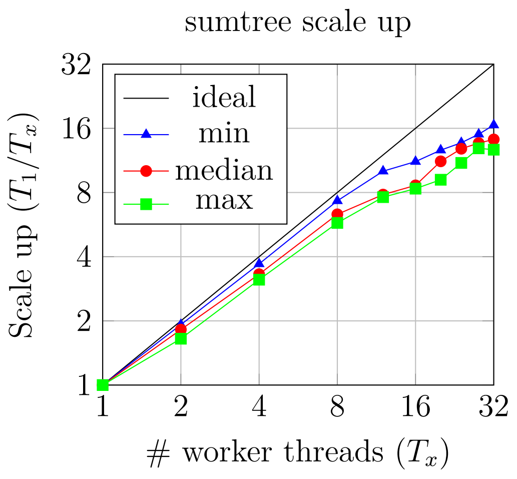
[3]Not counting fib without threshold since there is a better version of that algorithm.

stealing. By benchmarking on an unbalanced tree the effectiveness of the stealing mechanism can be demonstrated.

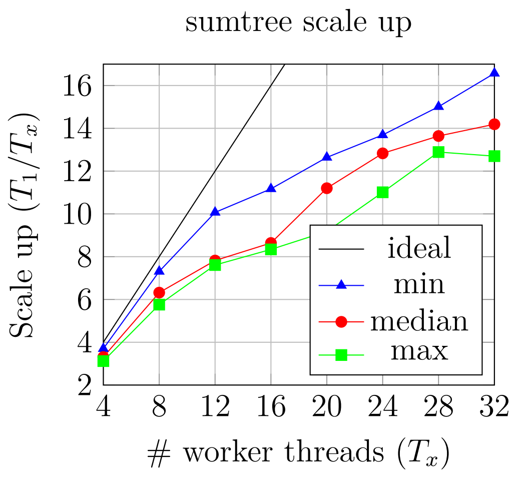
The code used to generate the trees that the benchmarks were run on can be viewed in appendix A.1.2. The trees generated from the function will have the property that for any given node with the children $C_1, ..., C_n$, child C_x will be a larger tree than C_{x-1}. Therefore, when all the n forks from the node have been stolen the thieves will have acquired different amount of work. This algorithm was chosen as it produces a tree somewhere between a balanced tree and a maximally unbalanced tree. The most unbalanced a tree can be is essentially the equivalent of a linked list, where every node has only one child. Such a list-like tree would create zero parallelism since every task, processing one node, would only create one subtask.

Different shapes of trees were benchmarked, but the differences were not significant enough to include here as separate benchmarks. A balanced tree with the same amount of nodes was slightly faster with a work overhead of 23.6 and a T_1/T_{32} scale up of 16.6. The list-like tree could not be tested as it made the worker thread overflow its stack because of the immense depth needed to create a tree of the same size as the other algorithms.

The unbalanced tree of size parameter 23 generated for benchmarks used 564 MiB of memory. The tree was generated once and the same instance was used for all benchmarks since is was being used read-only.



(a) Log-log presentation of all measured values.



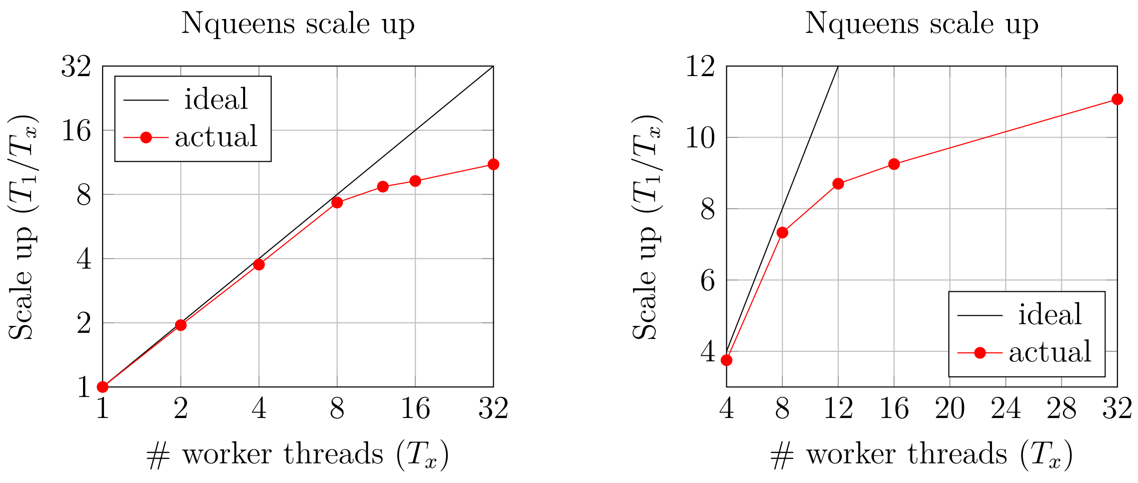
(b) Linear presentation of the part diverging from the ideal.

Figure 4.3: The scaling of computation time of the sumtree algorithm as more worker threads are added. The three lines represent the scale up for the fastest, median and slowest sample in each benchmark.

The benchmarked samples collected for sumtree are more spread out than for the other algorithms. For some benchmarks the fastest and slowest samples were more than 20% away from the median. Because of the spread out results, the scale up graphs for sumtree in figure 4.3 show not only the median time, but also the min and max. All min, max and median sample times are presented in table A.2.2 in the appendix.

4.6.3 Nqueens

One of the initial reasons for including nqueens in the benchmarking suite was to compare it with the performance presented in the Cilk paper [8]. In the Cilk 5 paper a work overhead of 0.99 and a scale up of 8.0 for 8 processors is presented. In the ForkJoin implementation a work overhead of 1.13 and scale up, relative serial execution, of 6.49 can be observed from table 4.1. Even if the results on ForkJoin are not as good as the ones in Cilk they are not far behind and they are among the best presented in this paper, only fib scales better. Also, as described in section 3.6.4, the implementations are not really comparable since they perform different work.



(a) Log-log presentation of all measured values.

(b) Linear presentation of the part diverging from the ideal.

Figure 4.4: Graph showing how the computation time of the nqueens algorithm scales as more worker threads are added.

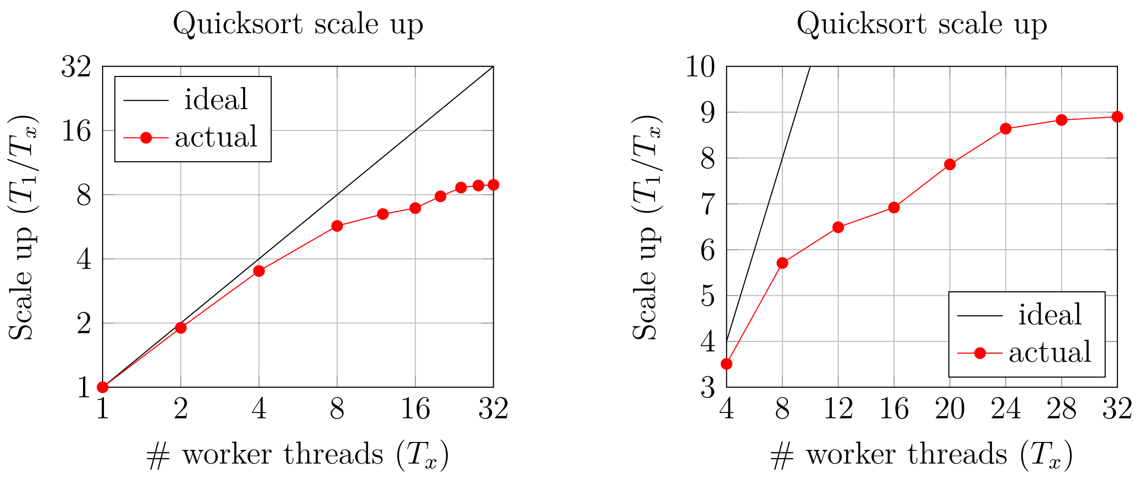
4.6.4 Quicksort

The input data for the quicksort benchmarks was an array with 10 million elements of unsorted 64 bit integers. The data was generated by a simple pseudorandom number generator with a fixed seed. The vector occupied a total of 76 MiB of memory.

Quicksort was the algorithm, among all the tested ones, that had the lowest work overhead of 1.01. In these benchmarks the algorithm was implemented with a granularity threshold of 1000 elements, meaning any input smaller than that would execute the serial version of the code. Implementing a granularity threshold is just as important in quicksort as in most other algorithms, without it the work overhead would be greatly increased.

How the quicksort algorithm scales with added worker threads can be observed in figure 4.5. The graphs show that the algorithm scales almost linearly up to 4 worker threads, after that the curve flattens out, but continues to slowly improve as more threads are added.

It was possible to see the effect of the quicksort problem described in section 3.6.5 from the sleep statistics in the benchmarks. The statics show that during the execution of the algorithm almost all threads sleep a considerable amount of time. When executing on 32 worker threads the threads slept, on average, 50 out of the 170 ms that the algorithm took to run. 20 ms of the time spent sleeping was before the first task was stolen, indicating that the algorithm is not able to split into subtasks directly.



(a) Log-log presentation of all measured values.

(b) Linear presentation of the part diverging from the ideal.

Figure 4.5: Graphs over how the computation time of the quicksort algorithm scales as more worker threads are added.

The results for quicksort can be compared with the results for *cilksort* in the Cilk paper [8]. The cilksort implementation is not completely accurate as the paper states that their serial version of sorting is a normal in-place quicksort, while the parallel version is a variation of the mergesort algorithm. Cilksort has a work overhead of 1.21, a T_1/T_8 scale up of 6.0 and a T_S/T_8 scale up of 5.0. Cilk's work overhead is slightly higher than ForkJoin's but their scale up relative to T_1 is slightly better. However, Cilk's scale up relative T_S is slightly worse than quicksort in ForkJoin.

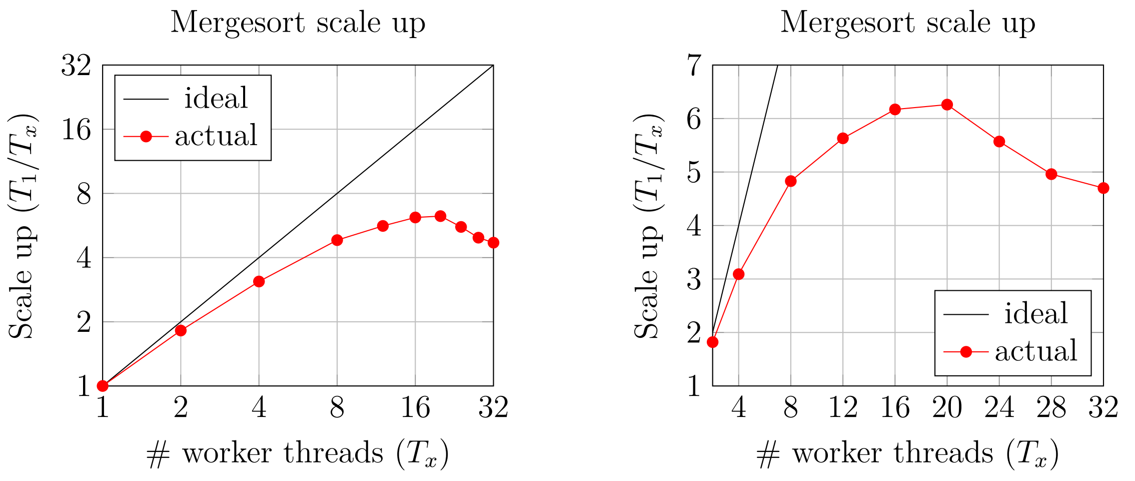
4.6.5 Mergesort

Mergesort was tested on the same input data as described for quicksort, a pseudorandomly generated vector of positive integers. It was also of the same size, 10 million elements.

The mergesort algorithm reaches peak performance when executed on 20 worker threads, where it has a scale up of approximately 6.3, and then performance goes down as more threads are added. This behavior has been observed during many different runs of the algorithm and with different sizes of input arrays. With smaller arrays the algorithm peaks on fewer threads, but it seems to always have optimal performance in the range of 15-20 threads. It is not only that the execution time levels off and stop improving, table A.10 in the appendix shows that the execution time increases after 20 threads.

The work overhead of mergesort is 1.13, which is much higher than for quicksort, and it is also among the highest overheads of all tested algorithms[4]. One factor that is known to affect work overhead is the granularity threshold, and that might have caused this result. Only one threshold was tested for mergesort, 1000 elements, since that was used for quicksort and using another threshold would have made comparisons to quicksort unfair.

It was possible to see the effect of the mergesort problem described in section 3.6.5 from the sleep statistics in the benchmarks. The statistics show that all threads were able to steal a task and start working instantly. Yet, out of the approximately 300 ms it took to run mergesort on 32 threads, all except one thread slept around 100 ms on average. If all threads had work instantly in the beginning but still had to sleep for one third of the running time, it is an indication that the algorithm have some serial parts that cannot be parallelized.



(a) Log-log presentation of all measured values.

(b) Linear presentation of the part diverging from the ideal.

Figure 4.6: Graphs over how the computation time of the mergesort algorithm scales as more worker threads are added.

[4]Not counting fib without threshold and sumtree since they are known to have bad overhead because of not implementing a granularity threshold

5 Discussion and conclusions

This thesis has presented a fully working work stealing fork-join library in the Rust language with code examples and benchmark results of six algorithms using the library. The provided API is not as flexible as it was originally planned, but it is fully useable for most use cases where a divide and conquer style algorithm is, or can be, implemented.

Since there is no silver bullet for parallelism in general, specialized tools need to exist for every parallelization style and ForkJoin meets this need and provides Rust with one such tool. More importantly than the library itself though, this thesis explores the possibilities, and discusses the implications of applying a well known algorithm to Rust's rather special memory model.

Even though the original plan was to provide one thread pool that could run tasks of any type signature, the existing implementation is definitely cleaner than the code required for a pool accepting any type. To make one instance of a `ForkPool` work for any type of task, all generics in the `WorkerThread` need to be removed and replaced with some fixed type working similar to a void pointer.

Every user have to judge for themselves if the ForkJoin library is easy to use or not. This report shows example algorithms and that it is possible to implement them in a way not very much longer or more complex than their serial counterparts.

When it comes to the three layer cake and the goal to fill one of the missing layers for Rust, the ForkJoin library does a good job. The library can be used under a layer of message passing, allowing components in a message passing system to spawn and maintain `ForkPool`s and perform calculations on them. Within the tasks of the algorithms using ForkJoin it should be possible to use SIMD calculations. However, this is not something that has been verified during this project since SIMD was out of the scope.

An example of a full stack usage of the three layer cake that should be possible, provided a functional SIMD library is found, is image brightness adjustments. The example system would be a message passing application where one component waited for messages containing images and a brightness scale value. The component send the image on to a `ForkPool` through an `AlgoOnPool` for brightness adjustments. The fork-join algorithm can then split up the images into parts and fork. When an image block is small enough, the desired operation can be applied to that image block with SIMD instructions, as adjusting brightness can be done by multiplying every pixel value with the scaling value. When the entire operation is done a message with the new image can be sent to some other component in the application.

5.1 Benchmark results

There are many factors that affect the timing results of executed code. Some of these factors stem from sources not controlled by the benchmark itself, such as outside computer load[1] and unlucky cache behavior. Both of these problems were minimized by running the benchmarks on a server with few other processes, and by collecting multiple samples of the same code execution.

One other factor that can greatly affect performance is the rust compiler and the implementation of classes and functions in the standard library. Since this report states exactly which version of Rust that is used, and the same version is used for all benchmarks, comparing the benchmarks with each other is accurate. Comparing benchmarks with other implementations in Rust or other languages might be possible to get an idea of relative performance, but should not be considered completely accurate.

5.1.1 Fibonacci and reduction of trees

The numbers in table 4.1 clearly shows that implementing a granularity threshold[2] is vital for acceptable performance in algorithms that otherwise fork to tasks so small that their amount of work is just a few instructions.

For algorithms that would fork to minuscule tasks, but where no granularity threshold can be accurately set without knowing characteristics of the input data, it seems to be best to leave them as serial implementations. Specialized parallel implementations can be written for scenarios where the input data always or usually have some known property.

The reason that the samples for sumtree were more spread out than for the other algorithms might be because of the large data structure that was involved, and page faults might be to blame. As stated in section 4.6.2, one tree instance used in the benchmarks occupied 564 MiB of memory. This can be compared with the sorting algorithms, which according to section 4.6.4 only used 76 MiB memory for their arrays of 10 million elements.

The performance of fib in ForkJoin seems low when compared with the results in Cilk. However, Cilk is a library developed by multiple people over many years. Also, Cilk is written directly in C and the paper reports a cost of spawning a task as being as low as 2-6 times the cost of an ordinary C function call. With that in mind about Cilk, the results in this report can be regarded as fully acceptable.

[1]Other processes on the same computer that perform tasks and occupy the CPU.

[2]See section 3.6.1

5.1.2 Nqueens

It would have been interesting to have a working abort mechanism in the ForkJoin library to test this algorithm with. It is likely that finding only the first valid placement of queens have considerably different performance characteristics than that of finding the complete set of solutions.

5.1.3 Sorting

As described in section 3.6.5, both quicksort and mergesort have some problems with scaling that does not depend on the fork-join implementation used. Considering this limitation in the algorithms, the presented performance results are expected and can be considered good.

The fact that the performance of mergesort decreases as more than 20 worker threads are added is interesting, but the reason is still unknown. The sleep data has been studied without finding a clue, all parts of the algorithm just take longer to execute on 32 threads than on 20 threads. The valgrind [24] tool cachegrind has been used to simulate cache behavior to examine if the slowdown was because of cache misses, this does not seem to be the case as no differences in cache misses were reported by the tool when analyzing mergesort on 20 and 32 threads. One reason could be that the OS scheduling overhead increase, but since the other algorithms scale to over 20 threads without a problem this is not likely.

6
Future work

This project has been very interesting and educational. Given my personal interest in both Rust and this specific library, I will most definitely continue to develop the library. I have discussed with my industry advisor, Lars Bergstrom who works at Mozilla, about the future of the library. We have agreed that a post about the library should be published on Mozilla's research blog and that he will introduce the library to the project manager of Rust, Nicholas Matsakis, for feedback.

A wish and challenge from Bergstrom is that Servo [21] is migrated to use ForkJoin rather than their built in implementation[1]. Making such migration work would provide good proof that the ForkJoin library is useable in real world applications.

There are possible optimizations to implement in the library inspired by how the worker threads in Cilk operate. Future work could include trying to lower the work overhead and increase scale up performance of using the ForkJoin library by implementing the optimizations from other libraries.

As described in section 3.5, the current API design is lacking some of the desired features. Work can be done to try to implement the API as desired, yielding a library that can be used easier and without the need to use unsafe for certain algorithm implementations.

To support all layers in the three layer cake and verify that they work together is important future work. This thesis builds partly on the ground that the three layer cake is a good approach for Rust as a language, but no work has been done to verify this and evaluate how well it fits together. Future work here is to find an existing, or create a new, SIMD implementation and then verify that the three layers, message passing, fork-join and SIMD work together as described in the three layer cake paper.

[1]See section 3.1.1

Bibliography

[1] Herb Sutter. “The Concurrency Revolution”. In: *C/C++ Users Journal* (Feb. 2005).

[2] Herb Sutter and James Larus. “Software and the Concurrency Revolution”. In: *Queue* 3.7 (Sept. 2005), pp. 54–62. ISSN: 1542-7730. DOI: `10.1145/1095408.1095421`.

[3] Arch D. Robison and Ralph E. Johnson. “Three Layer Cake for Shared-memory Programming”. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ParaPLoP ’10. Carefree, Arizona: ACM, 2010, 5:1–5:8. ISBN: 978-1-4503-0127-5. DOI: `10.1145/1953611.1953616`. URL: `http://doi.acm.org/10.1145/1953611.1953616`.

[4] Mozilla corporation. *The Rust Programming Language*. `http://www.rust-lang.org/`. June 2015.

[5] Nicholas Matsakis. *Data Parallelism in Rust*. `http://smallcultfollowing.com/babysteps/blog/2013/06/11/data-parallelism-in-rust/`. Blog. Blog, Accessed: 2015-02-11. 2013.

[6] F. Warren Burton and M. Ronan Sleep. “Executing Functional Programs on a Virtual Tree of Processors”. In: *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*. FPCA ’81. Portsmouth, New Hampshire, USA: ACM, 1981, pp. 187–194. ISBN: 0-89791-060-5. DOI: `10.1145/800223.806778`. URL: `http://doi.acm.org/10.1145/800223.806778`.

[7] Robert D. Blumofe et al. “Cilk: An Efficient Multithreaded Runtime System”. In: *Journal of Parallel and Distributed Computing* 37.1 (Aug. 1996). (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’95)*, pages 207–216, Santa Barbara, California, July 1995.), pp. 55–69. URL: `ftp://theory.lcs.mit.edu/pub/cilk/cilkjpdc96.ps.gz`.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In: *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation (PLDI)*. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998. Montreal, Quebec, Canada, June 1998, pp. 212–223.

[9] Simon Marlow, Ryan Newton, and Simon P. Jones. “A monad for deterministic parallelism”. In: *Haskell Symposium*. Tokyo, 2011.

[10] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. “The Data Locality of Work Stealing”. English. In: *Theory of Computing Systems* 35.3 (2002), pp. 321–347. ISSN: 1432-4350. DOI: 10.1007/s00224-002-1057-3. URL: http://dx.doi.org/10.1007/s00224-002-1057-3.

[11] Michael A. Rainey. “Effective scheduling techniques for high-level parallel programming languages”. PhD thesis. University of Chicago, Aug. 2010.

[12] Doug Lea. “A Java Fork/Join Framework”. In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA ’00. San Francisco, California, USA: ACM, 2000, pp. 36–43. ISBN: 1-58113-288-3. DOI: 10.1145/337449.337465. URL: http://doi.acm.org/10.1145/337449.337465.

[13] 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. Las Vegas, Nevada, USA: ACM, 2005, pp. 21–28. ISBN: 1-58113-986-1. DOI: 10.1145/1073970.1073974. URL: http://doi.acm.org/10.1145/1073970.1073974.

[14] OpenMP Architecture Review Board. *OpenMP*. http://openmp.org/wp/. June 2015.

[15] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Tech. rep. July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[16] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*. 1994, pp. 356–368.

[17] Nicholas Matsakis. *Guaranteeing memory safety in Rust*. https://air.mozilla.org/guaranteeing-memory-safety-in-rust/. Video presentation. Video presentation, Accessed: 2015-05-25. 2014.

[18] Eric Reed. *Patina: A Formalization of the Rust Programming Language*. Tech. rep. University of Washington, Feb. 2015. URL: ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf.

[19] Mozilla Corporation. *Rust documentation on ownership*. http://doc.rust-lang.org/nightly/book/ownership.html. June 2015.

[20] Linus Färnstrand. *ForkJoin*. https://github.com/faern/forkjoin. June 2015.

[21] Mozilla Corporation. *The Servo Browser Engine*. https://github.com/servo/servo. June 2015.

[22] Jorge Aparicio. *Git repository for Criterion.rs*. https://github.com/japaric/criterion.rs. June 2015.

[23] Bryan O’Sullivan. *Official criterion homepage*. http://www.serpentine.com/criterion/. June 2015.

[24] Julian Seward. *Valgrind’s cachegrind manual*. http://valgrind.org/docs/manual/cg-manual.html. June 2015.

A

Appendix 1

A.1 Code used for benchmarks

A.1.1 Fibonacci

```rust
/// Benchmarking method used in Criterion.rs for parallel fib
pub fn parfib(b: &mut Bencher, threads: usize, &i: &usize) {
    let forkpool = ForkPool::with_threads(threads);
    let fibpool = forkpool.init_algorithm(FIB);

    b.iter_with_large_drop(|| {
        let job = fibpool.schedule(test::black_box(i));
        job.recv().unwrap()
    })
}

/// Constants for defining the two algorithms
const FIB: Algorithm<usize, usize> = Algorithm {
    fun: fib_task,
    style: AlgoStyle::Reduce(ReduceStyle::NoArg(fib_join)),
};
const FIB_NO_THRESHOLD: Algorithm<usize, usize> = Algorithm {
    fun: fib_task_no_threshold,
    style: AlgoStyle::Reduce(ReduceStyle::NoArg(fib_join)),
};

/// Task executed for fib with a threshold
fn fib_task(n: usize) -> TaskResult<usize, usize> {
    if n <= 20 {
        TaskResult::Done(fib(n))
    } else {
        TaskResult::Fork(vec![n-2,n-1], None)
    }
}
```

```
/// Task executed for fib without a threshold
fn fib_task_no_threshold(n: usize) -> TaskResult<usize, usize> {
    if n < 2 {
        TaskResult::Done(1)
    } else {
        TaskResult::Fork(vec![n-2,n-1], None)
    }
}

/// A common join that simply sums up all the results
fn fib_join(values: &[usize]) -> usize {
    values.iter().fold(0, |acc, &v| acc + v)
}

/// The sequential implementation of fib for comparison
/// and use after granularity threshold has been reached
fn fib(n: usize) -> usize {
    if n < 2 {
        1
    } else {
        fib(n-1) + fib(n-2)
    }
}
```

A.1.2 Reduction of trees

The algorithm for reducing trees is called `sumtree` and just summarize one value from each node in the tree.

```
/// Definition of the tree structure used.
/// A node can have arbitrarily many children
pub struct Tree {
    value: usize,
    children: Vec<Tree>,
}

/// Benchmarking method used in Criterion.rs for parallel sumtree
pub fn par_sumtree(b: &mut Bencher, threads: usize, tree: &Tree) {
    let forkpool = ForkPool::with_threads(threads);
    let sumpool = forkpool.init_algorithm(Algorithm {
        fun: sum_tree_task,
        style: AlgoStyle::Reduce(ReduceStyle::Arg(sum_tree_join)),
    });

    b.iter(|| {
        let job = sumpool.schedule(test::black_box(tree));
        job.recv().unwrap()
    });
}

/// Task executed for sumtree
fn sum_tree_task(t: &Tree) -> TaskResult<&Tree, usize> {
    if t.children.is_empty() {
        TaskResult::Done(t.value)
    } else {
        /// Collect a vector of references to all child nodes.
        let mut fork_args: Vec<&Tree> = vec![];
        for c in t.children.iter() {
            fork_args.push(c);
        }
        TaskResult::Fork(fork_args, Some(t.value))
    }
}

/// Join function summing all the values together
fn sum_tree_join(value: &usize, values: &[usize]) -> usize {
    *value + values.iter().fold(0, |acc, &v| acc + v)
}

/// Sequential implementation of sumtree
fn sum_tree_seq(t: &Tree) -> usize {
    t.value + t.children.iter().fold(0, |acc, t2| acc + sum_tree_seq(t2))
}

/// The algorithm used to generate an unbalanced tree.
fn gen_unbalanced_tree(depth: usize) -> Tree {
    let mut children = vec![];
    for i in 0..depth {
        children.push(gen_unbalanced_tree(i));
    }
    Tree {
        value: depth + 1000,
        children: children,
    }
}
```

A.1.3 Nqueens

```
pub type Queen = usize;
pub type Board = Vec<Queen>;
pub type Solutions = Vec<Board>;
```

```
const NQUEENS_SEARCH: Algorithm<(Board,usize), Board> = Algorithm {
    fun: nqueens_task_search,
    style: AlgoStyle::Search,
};

/// Task function for parallel nqueens using ‘search style‘
fn nqueens_task_search((q, n): (Board, usize)) ->
    TaskResult<(Board,usize), Board>
{
    if q.len() == n {
        TaskResult::Done(q)
    } else {
        let mut fork_args: Vec<(Board, usize)> = vec![];
        for i in 0..n {
            let mut q2 = q.clone();
            q2.push(i);

            if ok(&q2[..]) {
                fork_args.push((q2, n));
            }
        }
        TaskResult::Fork(fork_args, None)
    }
}

/// Sequential implementation of nqueens.
/// This is the serial version of the ‘reduce style‘ since
/// everything that was measured anyway was the time it took to
/// collect every possible solution.
fn nqueens_reduce(q: &[Queen], n: usize) -> Solutions {
    if q.len() == n {
        return vec![q.to_vec()];
    }
    let mut solutions: Solutions = vec![];
    for i in 0..n {
        let mut q2 = q.to_vec();
        q2.push(i);
        let new_q = &q2[..];

        if ok(new_q) {
            let more_solutions = nqueens_reduce(new_q, n);
            solutions.push_all(&more_solutions[..]);
        }
    }
    solutions
}
```

```rust
/// Helper function checking if a placement of queens is ok or not
fn ok(q: &[usize]) -> bool {
    for (x1, &y1) in q.iter().enumerate() {
        for (x2, &y2) in q.iter().enumerate() {
            if x2 > x1 {
                let xd = x2-x1;
                if y1 == y2
                   || y1 == y2 + xd
                   || (y2 >= xd && y1 == y2 - xd)
                {
                    return false;
                }
            }
        }
    }
    true
}
```

A.1.4 Quicksort

```rust
/// Quicksort task, drops to sequential operation on arrays smaller
/// than or equal to 1000 elements
fn quicksort_task(d: &mut [usize]) -> TaskResult<&mut [usize], ()> {
    let len = d.len();
    if len <= 1000 {
        quicksort_seq(d);
        TaskResult::Done(())
    } else {
        let pivot = partition(d);
        let (low, tmp) = d.split_at_mut(pivot);
        let (_, high) = tmp.split_at_mut(1);

        TaskResult::Fork(vec![low, high], None)
    }
}

/// Join function doing nothing.
fn quicksort_join(_: &[()]) -> () {}

/// Sequential quicksort
pub fn quicksort_seq(d: &mut [usize]) {
    if d.len() > 1 {
        let pivot = partition(d);

        let (low, tmp) = d.split_at_mut(pivot);
        let (_, high) = tmp.split_at_mut(1);

        quicksort_seq(low);
        quicksort_seq(high);
    }
}

/// Partition function
fn partition(d: &mut[usize]) -> usize {
    let last = d.len()-1;
    let pi = pick_pivot(d);
    let pv = d[pi];
    d.swap(pi, last); // Put pivot last
    let mut store = 0;
    for i in 0..last {
        if d[i] <= pv {
            d.swap(i, store);
            store += 1;
        }
    }
    if d[store] > pv {
        d.swap(store, last);
        store
    } else {
        last
    }
}

/// Function picking pivot element as
/// the median of the first, middle and last element
fn pick_pivot(d: &[usize]) -> usize {
    let len = d.len();
    if len < 3 {
        0
    } else {
        let is = [0, len/2, len-1];
        let mut vs = [d[0], d[len/2], d[len-1]];
        vs.sort();
        for i in is.iter() {
            if d[*i] == vs[1] {
                return *i;
            }
        }
        unreachable!();
    }
}
```

A.1.5 Mergesort

```
fn parallel_mergesort(threads: usize, data: &mut [usize]) {
    let forkpool = ForkPool::with_threads(threads);
    let sortpool = forkpool.init_algorithm(Algorithm {
        fun: mergesort_task,
        style: AlgoStyle::Reduce(ReduceStyle::NoArg(mergesort_join)),
    });
    let job = sortpool.schedule(data);
    job.recv().unwrap();
}

/// Parallel mergesort task.
/// Switches over to sequential quicksort on arrays
/// smaller than or equal to 1000 elements.
fn mergesort_task(d: &mut [usize]) ->
    TaskResult<&mut [usize], (Unique<usize>, usize)>
{
    let len = d.len();
    if len <= 1000 {
        quicksort_seq(d);
        TaskResult::Done(unsafe{(Unique::new(d.as_mut_ptr()), len)})
    } else {
        let (low, high) = d.split_at_mut(len / 2);
        TaskResult::Fork(vec![low, high], None)
    }
}

/// Join operation for the parallel mergesort.
/// Checks that two arrays are given and that they
/// are placed just beside each other.
fn mergesort_join(xs: &[(Unique<usize>, usize)]) -> (Unique<usize>, usize) {
    assert_eq!(2, xs.len());
    let (ref lowp, lowl) = xs[0];
    let (ref highp, highl) = xs[1];

    let low = unsafe { mem::transmute::<&[usize], &mut [usize]>(
        slice::from_raw_parts(**lowp, lowl)) };
    let high = unsafe { mem::transmute::<&[usize], &mut [usize]>(
        slice::from_raw_parts(**highp, highl)) };

    // Make sure that the given arrays are next to each other
    assert_eq!(unsafe { low.as_ptr().offset(low.len() as isize) },
        high.as_ptr());

    merge(low, high); // Perform the actual merge
    // Take out the length and an unsafe pointer to the result
    unsafe {
        let mut ret = Unique::new(mem::transmute(ptr::null::<usize>()));
        ptr::copy(lowp, &mut ret, 1);
        (ret, lowl + highl)
    }
}

/// Sequential implementation of mergesort
fn mergesort_seq(d: &mut [usize]) {
    let len = d.len();
    if len < 1000 {
        quicksort_seq(d);
    } else {
        let mid = len / 2;
        let (low, high) = d.split_at_mut(mid);

        mergesort_seq(low);
        mergesort_seq(high);
        merge(low, high);
    }
}

/// Allocates a buffer vector and merges the given
/// arrays into that buffer, in the end it copies the data
/// back to the original arrays
fn merge (xs1: &mut [usize], xs2: &mut [usize]) {
    let (len1, len2) = (xs1.len(), xs2.len());
    let len = len1+len2;
    let (mut il, mut ir) = (0, 0);

    let mut buf: Vec<usize> = Vec::with_capacity(len);

    for _ in 0..len {
        if il < len1 && (ir >= len2 || xs1[il] <= xs2[ir]) {
            buf.push(xs1[il]);
            il = il + 1;
        } else {
            buf.push(xs2[ir]);
            ir = ir + 1;
        }
    }
    unsafe {
        ptr::copy(buf.as_ptr(), xs1.as_mut_ptr(), len1);
        ptr::copy(buf.as_ptr().offset(len1 as isize),
            xs2.as_mut_ptr(), len2);
    }
}
```

A.2 Verbose benchmark results

In this section a violin plot of every benchmark will be included. The violin plots shows the distribution of benchmark times for all the samples in one benchmark. The plots illustrate how stable the benchmarks are and how close to each other the different samples are. A very wide violin shows that the different samples yielded quite different timings, and a narrow violin shows that all the samples ended up with approximately the same execution time. Inside each violin there is a black crossmark showing the median time for all samples in that benchmark.

The included tables contain both timing results and calculated values on these timings. The first five columns contain the fastest, slowest and median timing results, as well as the lower and upper quartile for the 20 samples. Rightmost in the tables are two columns labeled *mindev* and *maxdev* these show how much the *min* and *max* samples deviate from the *median*.

A.2.1 Fibonacci

# threads	*min*	Q_1	*median*	Q_3	*max*	*mindev*	*maxdev*
seq	1055	1061	1061	1064	1071	0.006	0.009
1	1072	1077	1079	1080	1084	0.006	0.005
2	548.7	557.0	558.6	560.5	561.8	0.018	0.006
4	282.5	283.1	283.3	283.9	290.6	0.003	0.026
8	141.3	141.8	142.1	142.6	151.2	0.006	0.064
12	103.3	108.5	109.9	111.0	115.0	0.060	0.046
16	76.28	86.92	92.79	94.48	97.30	0.178	0.049
20	60.68	67.44	73.89	77.60	79.82	0.179	0.080
24	56.59	56.85	58.19	60.41	68.25	0.027	0.173
28	52.89	53.04	53.31	55.00	56.06	0.008	0.052
32	49.74	49.99	50.73	52.33	64.43	0.020	0.270

Figure A.1: Timing results for benchmark on fib with argument 42 and a threshold of 20. All numbers are in milliseconds and rounded to 4 significant digits.

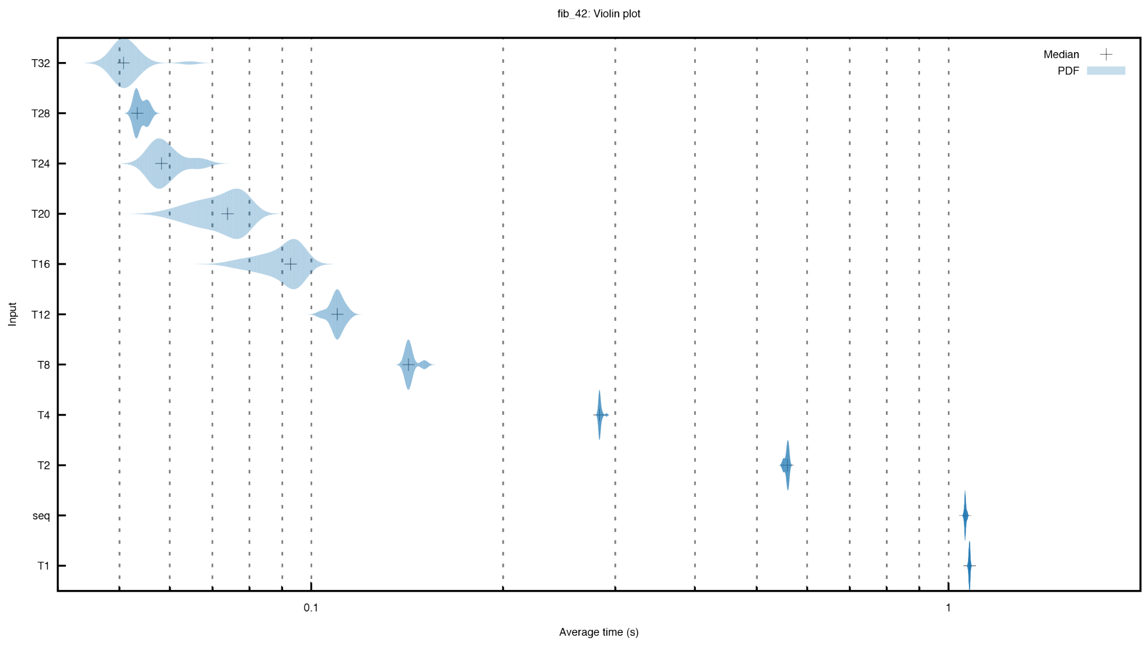


Figure A.2: Violin plot of fib with argument 42 and granularity threshold of 20.

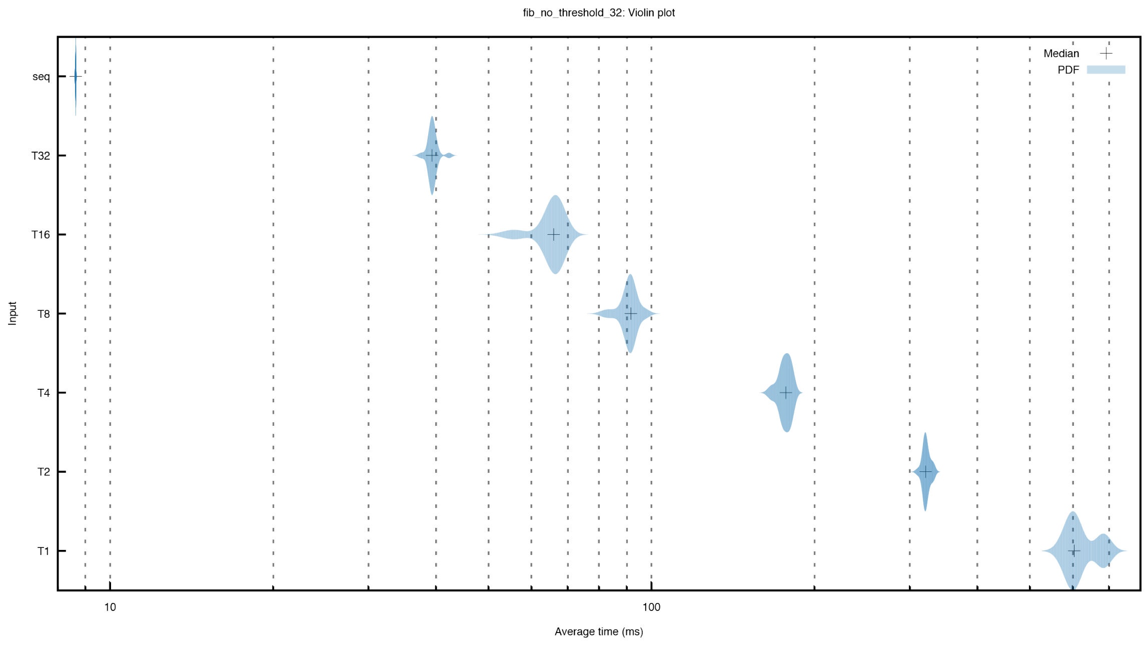


Figure A.3: Violin plot of fib with argument 32 and no threshold.

A.2.2 Reduction of trees

# threads	*min*	Q_1	*median*	Q_3	*max*	*mindev*	*maxdev*
seq	31.78	31.82	31.85	31.93	32.05	0.002	0.006
1	816.6	820.3	825.2	834.3	846.6	0.010	0.026
2	428.8	440.7	454.5	468.3	500.9	0.057	0.102
4	223.1	244.3	249.2	259.8	264.6	0.105	0.062
8	112.8	127.7	130.5	131.6	143.2	0.136	0.097
12	81.92	104.7	105.6	106.9	108.5	0.224	0.027
16	73.86	84.60	95.49	97.47	98.95	0.227	0.036
20	65.27	68.52	73.70	80.85	89.85	0.114	0.219
24	60.28	62.80	64.35	69.15	74.96	0.063	0.165
28	54.97	59.55	60.48	60.96	64.01	0.091	0.058
32	51.39	57.45	58.16	58.88	64.98	0.116	0.117

Figure A.4: Timing results for benchmark on sumtree with argument 23. All numbers are in milliseconds and rounded to 4 significant digits.

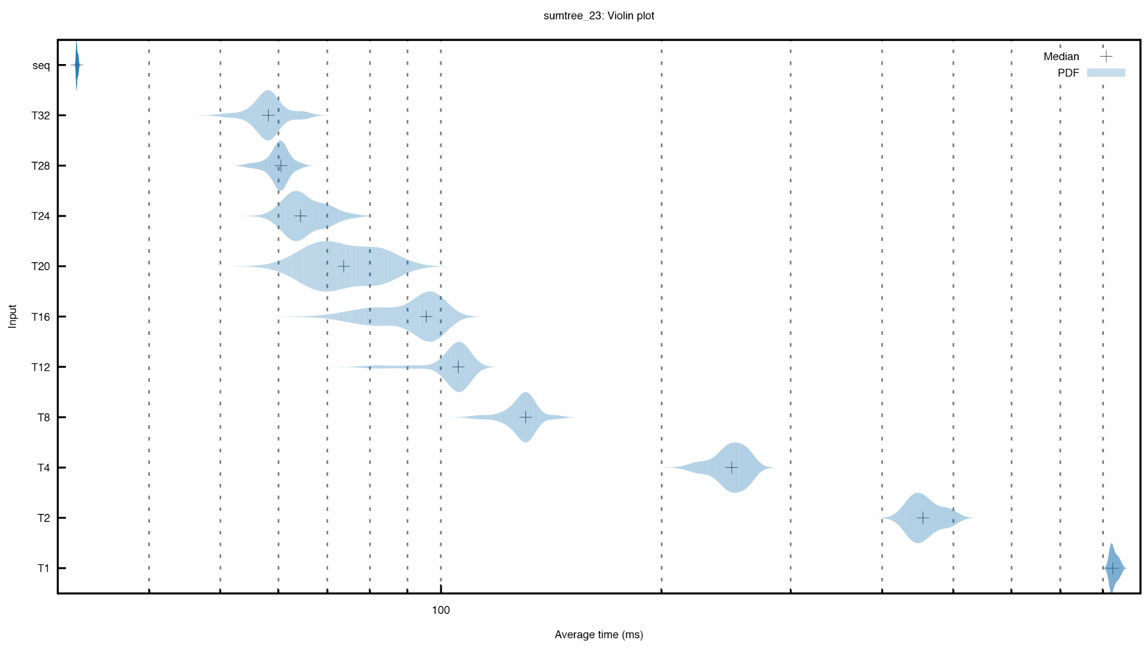


Figure A.5: Violin plot of sumtree with argument 23.

A.2.3 Nqueens

# threads	*min*	Q_1	*median*	Q_3	*max*	*mindev*	*maxdev*
seq	148.0	1493	1498	1502	1508	0.012	0.007
1	1654	1686	169.0	1696	1724	0.021	0.020
2	855.1	858.7	868.7	881.6	894.0	0.016	0.029
4	443.0	449.9	451.4	453.4	459.8	0.019	0.019
8	223.4	227.8	230.6	232.0	242.5	0.031	0.052
12	173.8	193.2	194.3	195.5	199.2	0.106	0.025
16	175.6	177.3	182.7	188.0	189.1	0.039	0.035
32	150.1	151.8	152.7	156.2	159.7	0.017	0.046

Figure A.6: Timing results for benchmark on nqueens with argument 12. All numbers are in milliseconds and rounded to 4 significant digits.

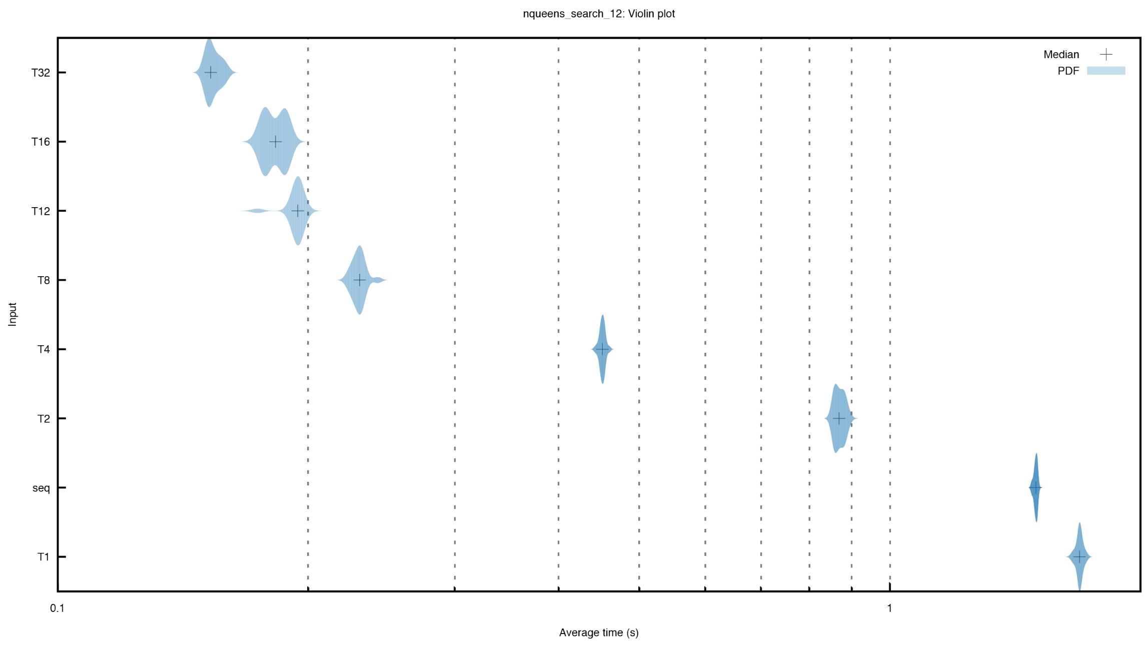


Figure A.7: Violin plot of nqueens with a 12x12 board.

A.2.4 Quicksort

# threads	*min*	Q_1	*median*	Q_3	*max*	*mindev*	*maxdev*
seq	1466	1472	1474	1476	1481	0.005	0.005
1	1464	1483	1486	1488	1493	0.015	0.005
2	770.5	778.1	782.7	786.9	790.3	0.016	0.010
4	420.9	422.4	423.6	424.6	427.0	0.006	0.008
8	253.5	259.8	260.6	261.9	265.0	0.027	0.017
12	215.8	225.1	229.0	230.0	232.5	0.058	0.015
16	197.8	208.9	214.7	218.0	223.6	0.079	0.041
20	176.8	185.3	189.0	191.7	202.2	0.065	0.070
24	165.6	170.5	172.1	173.3	186.2	0.038	0.082
28	163.6	167.6	168.4	170.1	175.1	0.029	0.040
32	163.7	166.1	167.0	168.3	173.3	0.020	0.038

Figure A.8: Timing results for benchmark on quicksort with 10 000 000 elements to sort. All numbers are in milliseconds and rounded to 4 significant digits.

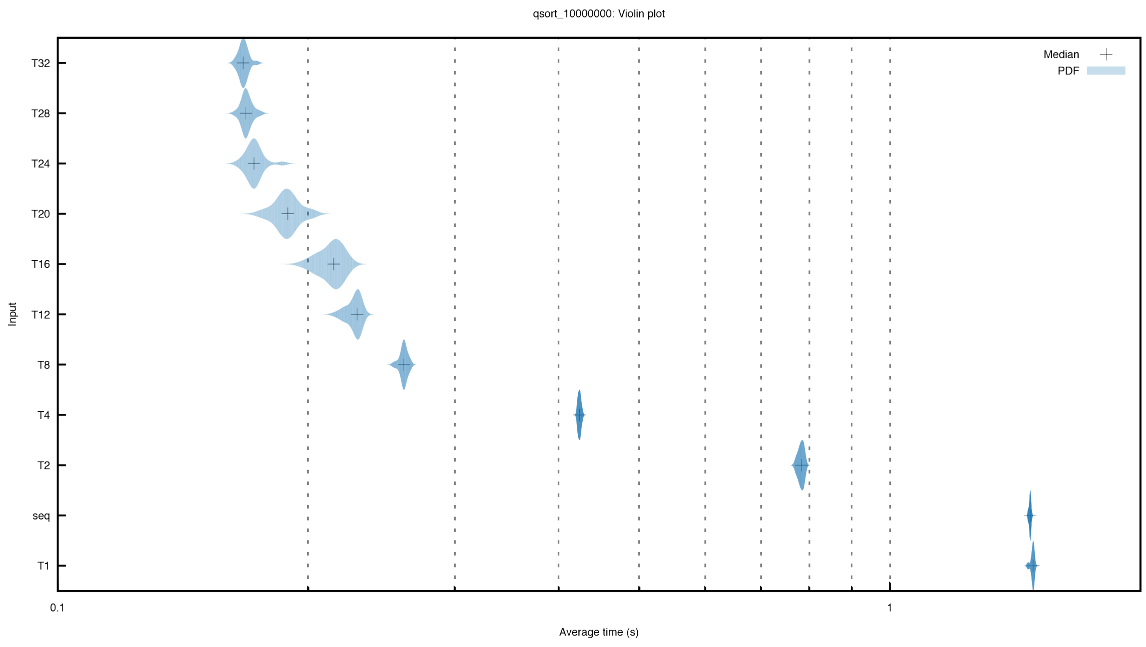


Figure A.9: Violin plot of quicksort with 10 000 000 elements and a granularity threshold of 1000 elements.

A.2.5 Mergesort

# threads	*min*	Q_1	*median*	Q_3	*max*	*mindev*	*maxdev*
seq	1233	1235	1237	1239	1240	0.003	0.002
1	1392	1394	1398	1402	1410	0.004	0.009
2	756.8	763.4	767.6	770.7	778.2	0.014	0.014
4	446.0	450.6	452.5	453.5	458.5	0.014	0.013
8	287.4	289.1	289.8	291.4	295.1	0.008	0.018
12	237.8	246.3	248.2	249.5	254.5	0.042	0.025
16	219.4	224.8	226.8	227.7	233.1	0.033	0.028
20	217.5	222.1	223.4	225.4	229.4	0.026	0.027
24	237.0	246.2	250.9	254.1	280.3	0.055	0.117
28	275.2	280.5	281.8	285.8	291.9	0.023	0.036
32	281.9	289.8	297.3	299.9	327.8	0.052	0.103

Figure A.10: Timing results for benchmark on mergesort with 10 000 000 elements to sort. All numbers are in milliseconds and rounded to 4 significant digits.

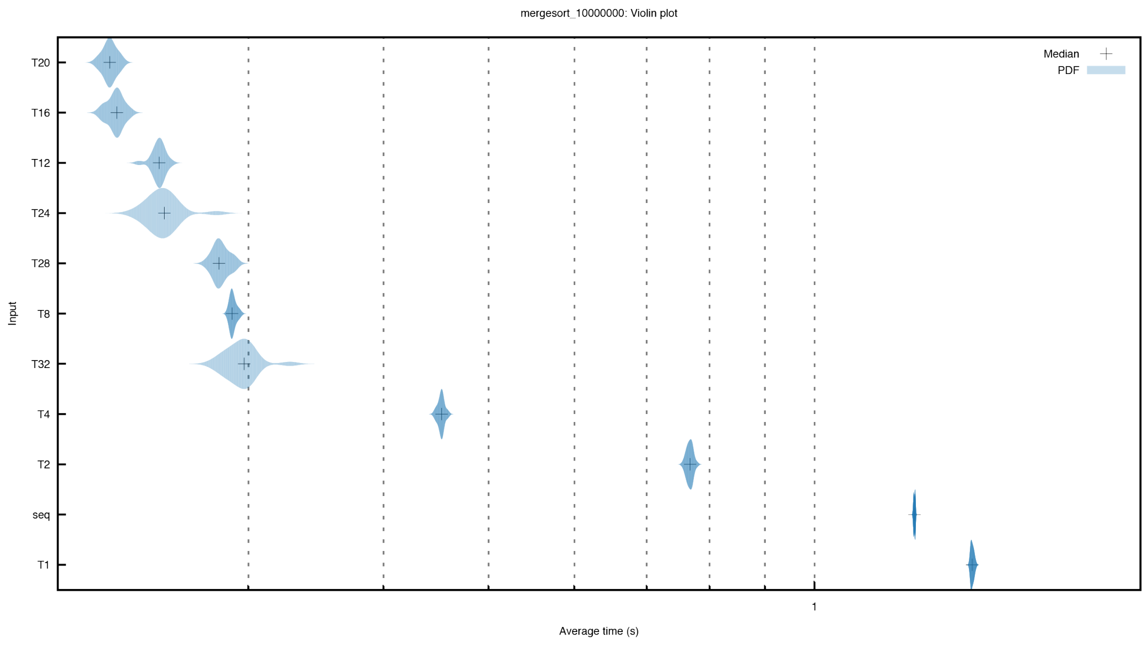


Figure A.11: Violin plot of mergesort with 10 000 000 elements and a granularity threshold of 1000 elements where it instead executes serial quicksort.