Need for Speed: Latency-Hiding Work-Stealing

Neil Weidinger

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

This skeleton demonstrates how to use the `infthesis` style for undergraduate dissertations in the School of Informatics. It also emphasises the page limit, and that you must not deviate from the required style. The file `skeleton.tex` generates this document and can be used as a starting point for your thesis. The abstract should summarise your report and fit in the space on the first page.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

This chapter provides a brief introduction and motivation to the problem of latency-hiding work stealing, as well as the aims of this project. An outline of the report is provided.

1.1 Motivation

Exploiting the parallel nature of modern processors is critical to achieving high performance. In an era where even entry-level consumer tier hardware features-shared memory parallelism, the need for software to be written that makes efficient use of these resources is key to unlocking efficiency gains. Unfortunately, the difficulty involved with the writing of parallel programs by explicitly specifying which computations should map to which processor is non-trivial and prone to introducing errors. Programmers need to manually schedule computations and enforce synchronization using low-level primitives such as locks and atomic operations, that are highly vulnerable to subtle and non-deterministic errors like race conditions and deadlocking.

This backdrop of the need to embrace parallelism has spurred the advancement of significant research and development in programming languages and paradigms to help assist writing such programs [32, 46]. One such paradigm is ***fork-join*** parallelism [18, 34], where programs can begin parallel execution at "fork" points and "join" to merge back and resume sequential execution again. This alleviates the programmer from having to manually managing parallel computation: one needs to only express the ***logical*** opportunities for ***possible*** parallelism, decoupling the programmer from the underlying runtime that takes care of handling scheduling and execution.

Such an underlying runtime to manage parallelism while the program executes must feature a ***scheduler*** to determine an efficient parallel execution. Runtime implementations such as Cilk [22], the Java Fork/Join framework [28], and TBB [1], commonly employ ***work-stealing*** [13]: a class of schedulers designed to deal with the problem of dynamically multithreaded computations on statically multithreaded hardware. In brief, work-stealing schedulers take care of scheduling and the accompanying issue of load balancing available work by using a fixed pool of worker threads that are each

responsible for maintaining a local deque to keep track of this work. Worker threads execute work from the bottom of their local deque, and when they run out of work they become ***thieves*** and ***steal*** from the top of the deques of other randomly selected worker threads.

Work-stealing has shown itself to be a very efficient approach to implementing fork-join parallelism. Such schedulers have strong theoretical performance guarantees [14] as well as proving themselves in practice [12]. Research on work-stealing has proven fruitful in the domain of traditional fine-grained parallelism, such as in high-performance and scientific computing where workloads are dominated by computational tasks: operations rarely incur latency and nearly all of the time a processor spends executing an operation is useful work. General modern workloads running on parallel architectures, however, frequently involve large degrees of operations that do incur latency, commonly in the form of I/O: waiting for user input, communicating to a remote client or server, dealing with hardware peripherals, etc. In such environments, classic work-stealing schedulers can suffer from large performance implications, depending on how much latency is incurred while executing the workload [32, 40, 46].

Classic work-stealing schedulers have no notion of latency-incurring operations nor incorporate them into the design of the scheduling algorithm. A worker thread that encounters a latency-incurring operation is ***blocked***: it is performing no useful work, and is simply wasting time waiting for the blocking operation to complete. Latency-incurring operations cause underutilization of the available hardware resources, potentially significantly impacting performance. An alternative to blocking operations are to use ***asynchronous*** (i.e. non-blocking) I/O operations, but those come with their own set of challenges of managing concurrent control flow [33, 42].

Under such workloads, large performance gains can be made from having either the latency-incurring operation cooperatively yield or the scheduler forcibly preempting the operation, and allowing another task that could be performing useful work to run instead. This allows the latency-incurring operation to wait out its latency in the background even while all hardware resources are being fully utilized on other tasks, thus ***hiding*** the latency. Singer et. al [41], introduce a latency-hiding work-stealing scheduling algorithm, ***ProWS***, that utilizes ***futures*** (a parallel language construct [23, 24]) to represent and schedule latency-incurring operations. Futures can be ***spawned*** to begin parallel execution, and return a handle future handle that can be ***touched*** (also commonly called ***await*** or ***get***) at a later point to retrieve the value of the operation. ProWS, by proactively stealing whenever the scheduling algorithm encounters a blocked future, can provide better bounds on execution time and other performance metrics than previous work [32, 43].

Scheduling futures is only one part of the equation to fully support latency-hiding work-stealing: a runtime system to efficiently dispatch and process latency-incurring operations is also necessary. The use of futures to hide latency is not much help if worker threads themselves must incur the cost of awaiting futures; dedicated I/O threads and integration with operating system event notification facilities is required. Singer et. al [40] present an extension of Cilk, called Cilk-L, that incorporates such runtime support with encouraging results.

TODO: better transition to talking about Rust

Rust [31] is a relatively new systems level programming language aimed at delivering the low-level abilities and performance characteristics of languages like C and C++, while ensuring far greater levels of memory and thread safety. It achieves this by using a rich static type system to allow the compiler to infer when and how values are safe to use, termed the "borrow checker". It also has first class support for asynchronous programming through the use of futures (futures in Rust have some idiosyncrasies described in section 2.3) and a quickly growing ecosystem surrounding asynchronous programming in Rust. Rayon [10, 2, 44] is a widely-used library level implementation for Rust programs (Cilk and its derivatives consist of a both Cilk to C level compiler and a runtime library) of work-stealing that supports task-level fork-join parallelism, but without latency-hiding.

Futures in Rust, unlike the hand-written futures used in Cilk-L, are easily composable: the compiler automatically generates a state machine that represents the current state of an asynchronous operation. This allows for ergonomic user-defined asynchronous operations that highly resemble regular sequential code, and powerful future combinators [4]. Supporting this behavior requires a few additional capabilities than what ProWS provides directly.

1.2 Aims

The aim of this project is to provide an implementation, building heavily upon the work of Singer et. al, of latency-hiding work-stealing that seamlessly integrates with the Rust language and ecosystem.

This report describes the design, implementation, and evaluation of ProWS-R, a variant of ProWS that is adapted for the particular futures found in Rust and its stringent safety guarantees.

This report describes ProWS-R, a variant of ProWS that is adapted for the particular futures found in Rust, as well as taking careful consideration with regards to the stringent safety guarantees and ergonomics of the language. The accompanying runtime library Rayon-LH, a fork of the Rayon library, is also presented. The design, implementation and subsequent evaluation of ProWS-R are ???

The explicit goals of this project are as follows:

- TODO: explicit goals
- Present a latency-hiding work-stealing algorithm, that takes accommodates the language-level futures construct in Rust
- Develop a prototype implementation of

1.3 Contributions

1.4 Report Outline

Chapter 2

Background: What Andy Giveth, Bill Taketh

Describe and outline background chapter.

2.1 Modern Computer Architecture

Ever since the advent of general-purpose microprocessor based computer systems, transistor density has been and continues to double roughly every two years. Famously known as Moore's law, for the first 30 years of the existence of the microprocessor the consequences of this graced the computing world with effortless biennial performance increases. Bestowed with this exponential growth of transistor density, chip designers could drastically increase core frequencies with each generation, and with ever increasing transistor budgets afford to design more complex architectural features like instruction pipelining, superscalar execution, and branch prediction. Without touching a line of code, software developers could expect programs to automatically double in performance every two years [25].

Accompanying Moore's law was another, related, effect: Dennard scaling. While Moore's law provides increased transistor counts, Dennard scaling allowed for this transistor doubling while ensuring power density was constant. The scaling law states roughly that as transistors get smaller, power density stays constant, meaning that power consumption with double the transistors stays the same. Additionally, as transistor sizes scale downward, the reduced physical distances enable reduced circuit delays, meaning an increase in clock frequency, boosting chip performance. When combined, with every technology generation transistor densities double, clock speeds increase by roughly 40%, and power consumption remains the same [16]. This remarkable scaling is what historically allowed for incredible performance gains year over year, all while keeping a reasonable energy envelope.

But starting around 2005, Dennard scaling has broken down: processors have reached the physical limits of power consumption in order to avoid thermal runaway effects that would require prohibitive cooling solutions (CPU chips that would melt would likely

be difficult to sell to customers). This is known as the power wall, and chip designers could no longer regularly rely on increasing clock frequencies to deliver performance gains [35]. The multicore era was born.

2.1.1 Rise of the Multicore Era

While the doubling of transistor count observation of Moore's law is still going strong, the historic predictable free performance lunch it became associated with is no longer what is once was. Instead of being dedicated to more complex architectural features in order to extract sequential performance on single core processors, the extra transistors are largely used to build more cores on a single processor. With diminishing returns on effort spent increasing single core performance, chip designers look to add multiple cores to be able to execute more instructions in parallel [36].

CPU performance can be described using the following equation [37], where performance is measured in terms of absolute execution time:

$$\text{CPU execution time for a program} = \frac{\text{Program instruction count} \cdot \text{CPI}}{\text{Clock rate (frequency)}}$$

where CPI is average clock cycles per instruction. No longer able to increase the clock frequency due to the power wall and increasing difficulty reducing CPI, efforts of hardware architects focused on simply reducing program instruction count per processor, by distributing instructions across multiple CPU cores to be executed in parallel [21].

Multicore processors are microprocessors containing multiple processors in a single integrated circuit, where each of these processors is known as a core. Almost all commodity multicore processors today are *symmetric multiprocessing (SMP)* systems, meaning all cores in a processor share the same physical address space. A single physical address space allows cores to operate on shared data.

Armed with multiple cores, different programs or different parts of the same program can be run at the same time in parallel, reducing the time required to perform the same amount of work on a single processor, boosting performance. No longer limited to a single processor core executing work, programs stand to drastically benefit in execution throughput by being run on multiple cores simultaneously.

As of 2021, it is difficult to find a processor that is not a multicore processor. The performance gains provided by having multiple cores have shown to be so profound that even the lowest end chips feature multiple cores. The Raspberry Pi Zero 2, a £13.50 board in the Raspberry Pi family of low cost single-board computers, features a 64-bit quad-core Arm Cortex-A53 CPU [30].

Initially, multicore processors may seem like a silver bullet to the question of what to do when faced with the power wall: for more performance, simply scale the number of cores! In reality, as is typical, the situation is more nuanced. Many workloads cannot be trivially diced up and processed on multiple cores, and even if so, support for splitting up work and then computing this work in parallel must be explicitly supported and designed for.

2.1.2 Parallel Computing and its Difficulties

Although multiple cores on a single chip running in parallel offer tantalizing performance benefits, there is one catch: programmers must write explicitly parallel programs. Software must be carefully designed such that it actually takes advantage of multiple processing units: a single-threaded application running on an 8 core CPU can only take advantage of 1/8 of such a chips potential throughput. Worst of all, after about nearly two decades of experience since the introduction of the first multicore processors, experience has shown that compared to traditional sequential programming, parallel programming is simply very difficult [20, 29, 36].

- There are many things a programmer must be aware of when writing parallel programs (**CONDENSE**)
  - Synchronization of memory accesses
  - Synchronization of control flow
  - Memory ordering
  - Race conditions
  - Deadlocks/livelocks
  - Lock-free programming and atomic operations
  - Workload partitioning
  - Workload balancing
  - Scheduling

Before the introduction of mainstream multicore processors, code was written with the intention of being executed in serial on a single core processor. Once CPU hardware started featuring multiple cores, programs did not magically rewrite themselves to take advantage of this increased firepower. Instead, developers had to manually identify the existing parallelism in their programs and refactor them to run as multiple computational threads [36].

Threads are an abstraction of units of scheduling and execution: the same program can have multiple threads of execution running concurrently. A ***thread scheduler*** manages these threads [1] and chooses when they should run on the CPU. Multicore processors allow threads to truly run in parallel, instead of just concurrently as would happen on a single core processor, as multiple cores can each be executing a thread simultaneously.

Threads are the most common approach to concurrent programming and by extension parallel programming, but are notoriously difficult to use efficiently and and ensure correctness with [29].

[1] Typical thread schedulers however have no knowledge of the thread workloads; all threads are opaque to the scheduler: the scheduler does not know whether a given thread has operations that must complete before operations in another thread, the scheduler simply chooses a thread to run according to its scheduling algorithm (that typically tries to schedule all threads fairly/evenly) and it is up to the programmer to explicitly program this thread synchronization.

Other approaches to parallel computing, for example network connected cluster based designs frequently found in the high performance computing (HPC) domain, will not be discussed in this report, although the ideas described could possibly be transferable.

Not only does software need to be written with the above concepts in mind, there is an inherent limit to how much a given program can even take advantage of the parallelism offered by multiple processors. Not all program can benefit from being run in parallel, and those that do have their maximum theoretical speedups capped by Amdahl's Law.

2.1.3 Amdahl's Law (REMOVE)

Amdahl's law provides an upper bound to the potential speedup a program can benefit from when the performance of a portion of the program is improved, and is often used in parallel computing to predict the maximum theoretical speedup when using multiple processors. A program that required T_{old} time to execute that now has a fraction α of it running in parallel on k processors has a theoretical speedup $S = T_{\text{old}}/T_{\text{new}}$ of

$$S = \frac{1}{(1-\alpha)+\alpha/k}$$

A derivation can be found in [17]. If we consider setting k to ∞, we can see the maximum possible speedup S_∞ of a program running on an infinite number of processors is $1/(1-\alpha)$. The major insight of Amdahl's law is that to speed up the entire program, the speed of a very large fraction of the program must be significantly improved, and even then there is an upper bound on the possible speedup that can be achieved. In other words, a program that wishes to take advantage of being run in parallel must exhibit an ample amount of parallelism (fraction of the program that can be run in parallel) to demonstrate an advantage when compared to running on a single processor.

2.2 Classical work stealing

2.2.1 Motivating Examples

- Fibonacci example for compute bound
- Include latency-incurring example here in BG chapter, or later?

2.2.2 DAG model of Parallel Computations

- Taken largely from [19, 26, 15]
- With typical sequential computing, all instructions can be defined as a totally ordered set of instructions, where the ordering specifies the execution order. With multithreaded computing, a computation can be defined as a partially ordered set of instructions, which may be executed in any order that complies with the partial ordering.
- This partial ordering in a multithreaded computation can be represented as a DAG $G = (V, E)$

- Each node in V represents an instruction to be executed
- Each directed edge in E represents dependencies between instructions
  - Edge $(u, v) \in E$ means that instruction u must execute before instruction v
  - A horizontal edge, called **continuation** edges, represents the sequential ordering in a given thread (threads are the horizontal shaded blocks of sequential instructions)
  - A thread can spawn another thread, which is indicated by a downward edge originating from a **spawn instruction**. Spawn instructions, the instruction that actually performs the spawn operation, are the only nodes with outdegree of 2. Spawns introduce parallelism: instructions in different threads can execute concurrently, and by extension execute in parallel.
  - An upward edge, called a **join edge**, represents points of synchronization: instruction nodes that have incoming join edges wait until all previous instructions have executed before proceeding.
- If a directed path exists between u and v, they are (logically) in series, and execute serially just like in typical serial programs (doesn't say anything about when they're executed, only that they must not execute in parallel and must execute in order specified)
- Otherwise if a path does not exist, they are (logically) in parallel, meaning they ***may*** execute in parallel (does not specify that they ***will*** execute in parallel, only that at runtime a scheduler is allowed to choose to run them in parallel by assigning them to available processors)

2.2.3 Analysis of Parallel Computations

- To analyze the theoretical performance of multithreaded program, we need a more formal way of describing the efficiency of such programs
- Let T_P be the running time of a multithreaded program run on P processors
- The **work** of a computation is the total time required to execute each computation node in the DAG. In other words, work is the time required to execute the computation on a single processor: T_1.
- The **span** (also called critical path) of a computation is the length of the longest sequence of computations that need to be executed serially, due to computation dependencies represented as edges in the DAG. In other words, the span is the running time if the computation could be run on an infinite number of processors, denoted by T_∞.
- Ideally, to reduce execution time, a computation minimizes the work and span
- Using the above two definitions, we arrive at two very useful results:
  - **Work law:** Since P processors can perform at most P operations in one time step, the total amount of work performed in T_P time is PT_P. Since the

total amount of work to be done is T_1, we see that

$$PT_P \geq T_1$$

This can also be interpreted as the fact that the time T_P to run the computation on P processors is at least the time taken to run on one processor T_1 divided by the number of processors P:

$$T_P \geq T_1/P$$

– **Span law:** Since the time taken to run a computation on a finite number of processors P cannot be faster than the time taken on an infinite number of processors, we have

$$T_P \geq T_\infty$$

- Now we can define a few useful performance metrics:
  - **Speedup:** Defined by the ratio $S_P = T_1/T_P$, expressing how much faster the computation is on P processors than on 1 processor. Rearranging the work law, we see that $T_1/T_P \leq P$, meaning that the speedup gained by running on P processors is at most P. When the speedup scales linearly with the number of processors $T_1/T_P = \Theta(P)$, we have ***linear speedup***, and when $T_1/T_P = P$ we have ***perfect linear speedup***.
  - **Parallelism:** The amount of ***parallelism*** in a computation is expressed by the ratio T_1/T_∞. This represents the average number of computations that can be performed in parallel at ***each step*** along the critical path. This is also the maximum possible speedup that can be achieved on any number of processors (using the span law: $T_1/T_P \leq T_1/T_\infty$). We see that there is not much point in using P processors when $P > T_1/T_\infty$, as the extra processors will just be idle not performing work.

2.2.4 Scheduling

- Our model of multithreaded computation does not specify which instructions to run on which processors at what point in time: this is the job of the **scheduler**.
- A scheduler constructs an **execution schedule** that maps instruction nodes in the multithreaded computation DAG to processors at each step in time [2]. A typical goal for a scheduler is to reduce absolute execution time (we later prove asymptotic bounds for our scheduler in terms of the work and span). At any given point in time a processor is either active or idle; a scheduler tries to minimize the amount of time a processor sits idle.

[2] In practice, our scheduler that breaks down the multithreaded DAG is a user-space scheduler that maps instructions on to threads (as many as there are processors), and an operating system level thread scheduler then schedules these OS level threads. In other words, a userspace scheduler maps instruction nodes onto a fixed number of OS threads, and the kernel level scheduler maps these threads onto hardware processor cores.

- An execution schedule must satisfy all constraints given by the edges present in the DAG, such that the partial ordering of instructions is satisfied.
- As mentioned in section 2.1.2, programmers can themselves schedule when threads in their programs should be run as well as ensure proper synchronization. This low-level manual thread orchestration, however, becomes increasingly difficult with the number of threads involved, especially when shared resources and thread synchronization are required: the sheer number of possible interleavings of code execution quickly becomes unwieldy. Not only are threads difficult to reason about conceptually, it is also important for the programmer to perform efficient load balancing of processors to optimize utilization of available computing resources. A programmer would have to manually use complex communication protocols to ensure each thread receives a balanced amount of work to execute, which can be difficult and error prone.
- An easier approach is to raise the level of abstraction, and instead only require programmers to merely ***expose*** sections of possible parallelism in their programs, while letting a runtime system take care of the thread creation and building an execution schedule. The runtime systems in turn then becomes responsible for such a schedule that not only satisfies all ordering constraints, but also achieves high performance and efficiency.

2.2.5 Work Stealing

2.3 Futures

TODO: put general stuff about futures concept here

2.3.1 Futures in Rust

TODO: describe how futures in Rust work here (e.g. state machine, async fns, executors, pinning, etc.)

2.3.2 Futures and DAGs

A diagram of a (partial) DAG for the distributed map and reduce example is shown in figure 2.1. The bolded red lines represent edges in the DAG that incur latency of the network request, that is simulated in the benchmark with a simple future that suspends for a desired duration. Without latency-hiding, *each* of the latency-incurring edges must be incurred by the worker threads, while with (perfect) latency-hiding just the cost of the *single* heaviest latency-incurring edge on the critical path is incurred. TODO: move this figure and explanation to the background, would provide a nice introduction and subsequent conclusion here

TODO: describe how futures can be incorporated into analysis of parallel computations using DAG model

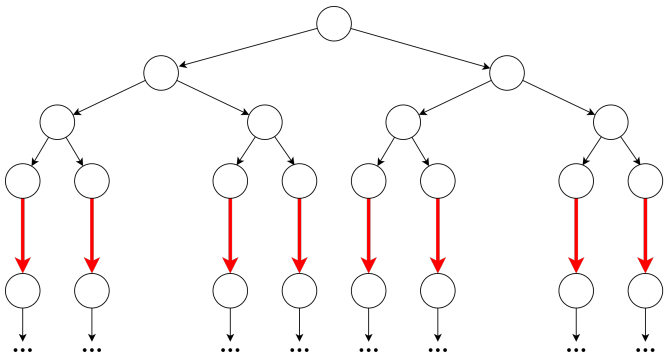

Figure 2.1: A partial DAG for map and reduce

2.4 Survey of Related Work

Chapter 3

Conceptual Latency-Hiding: To Wait Or Not to Wait?

This chapter describes the high-level overview of the ProWS-R latency-hiding work stealing algorithm, without diving into the details of the implementation. Section 3.1 provides an overview of the core scheduling algorithm, and the considerations that allow for repeatedly polled futures found in Rust. Section 3.2 describes the additional runtime support necessary to provide the latency-hiding capabilities of the scheduler.

3.1 The ProWS-R Algorithm

This section is heavily based on the work done in [41], where the authors introduce the ProWS scheduling algorithm. ProWS is a provably efficient algorithm that introduces support for the generalized concept of futures, that deviates from traditional work-stealing algorithms by being ***proactive***, detailed in section 3.1.1. Presented here is an exposition of the ProWS-R algorithm, a variant of ProWS, with details on the considerations taken to support the Rust implementation of futures.

3.1.1 Parsimonious vs Proactive Work-Stealing

Before jumping into the core algorithm, it's insightful to touch upon the differences between parsimonious and proactive work-stealing. The consequences of this mainly affect the theoretical execution bound (section 3.1.5) and on the number of ***deviations*** [43], a metric used to analyze the theoretical performance of parallel executions. Informally, the difference between a parsimonious and proactive work-stealing scheduler boil down to what actions are taken upon encountering a blocked future: a parsimonious scheduler continues execution by popping nodes off its worker deque, while a proactive scheduler opts to immediately become a thief and attempts to steal work from elsewhere.

The classic work-stealing scheduler, as described in section 2.2.5, is parsimonious. Given the online scheduling problem where the computational DAG unfolds as execution proceeds, it is the responsibility of the scheduler to map work to available

processing resources in a way that is efficient and still preserves the sequential dependencies of nodes in the DAG. Parsimonious scheduling achieves this by having each worker thread maintain its own deque of nodes that represent work to be executed, and having workers continuously pops nodes off and executing them. Upon completion of a node execution, the node may enable zero, one, or two child nodes. If zero nodes are enabled, it attempts to pop off its deque. If one node is enabled, it immediately executes the enabled node. If two nodes are enabled, it pushes one of the nodes to its end of the deque and executes the other.

Only when a worker runs out of work, signified by its deque being empty, does it attempt to steal work from another worker. It becomes a thief and randomly selects a victim deque, attempting to pop off a node from the top. Crucially, in the context of dealing with futures, a blocked future simply falls under the case of zero nodes, meaning a worker continues looking for work in its local deque. The scheduler presented by Muller and Acar [32] is such an algorithm: upon encountering a blocked future, it sets the suspended future to the side but continues executing nodes from its deque.

In contrast, the defining characteristic of proactive work-stealing is what occurs instead upon encountering a blocked future: the deque is suspended and the worker immediately attempts to find work elsewhere, by becoming a thief. In ProWS-R, a worker marks its current deque (that it popped the future off of) as ***suspended***, randomly selects another worker to assign this suspended deque to, and then tries to steal work from other workers. Importantly, this means that although there are P workers, there can be more than P deques at a given time. Although it may initially appear counterintuitive to proactively steal, as it may seem to increase the amount of steal attempts and corresponding scheduler overhead, but doing so provides a better bound on execution time given latency-incurring operations (section 3.1.5).

3.1.2 Algorithm Overview

Presented here is a description of the ProWS-R algorithm, the conceptual data structures used, and the adjustments made to accommodate the futures found in Rust. Again, this is largely based upon the work of Singer et. al, and their work should be consulted as reference.

The principle idea behind ProWS-R is that there can be multiple deques in the system at any given time, and each worker thread owns an ***active deque*** that they work off of. Whenever a worker thread encounters a blocked future, its current active deque is marked as suspended, the worker relinquishes ownership of the deque, and it attempts to find work elsewhere. The act of suspending deques allows for the latency of latency-incurring operations to be hidden while worker threads can fully utilize the available hardware resources to make progress on remaining available work. This differs from classic work-stealing, where such a scheduler without even the concept of latency-incurring operations would simply treat the operation as a regular computation, and be forced to block until the latency is incurred. When a worker thread is executing a node and does not encounter a blocked future, the algorithm proceeds the same as classic work stealing.

When a blocked future reaches completion, a callback is executed that marks the deque as ***Resumable***, indicating that the previously suspended deque now has work available and is free to have its work stolen by worker threads. Worker threads have the ability to either steal just the top node off of other deques (including the active deques of other workers), or ***mug*** entire deques that are marked as Resumable (but are not the active deques of other workers), and claim ownership of such deques. Mugging allows for the entire deque to be stolen in one go, as opposed to workers having to repeatedly steal nodes one-by-one off of such deques (since they're not the active deques of any other workers).

3.1.3 Data Structures

Deques

Like in classic work-stealing, nodes that represent work in the computational DAG are stored in deques. Deques are assumed to have support for concurrent operations. Each worker thread owns an active deque that they pop nodes off the bottom of and execute, like in classic work-stealing. If nodes spawn child nodes, these are pushed to the bottom of the deque. Worker threads, when stealing, pop nodes off the top of these deques. Worker threads also have the ability to steal entire deques at once (called mugging) that then become the new designated active deque for the respective worker thread. Each worker thread, in addition to having an active deque, manages a set of ***stealable*** deques, called a ***stealable set***, that are not being actively worked on but contain ready nodes that can be stolen and executed. Deque operations are assumed to take constant amortized time.

Deques support the following operations:

- `popTop`: pop top node off of deque
- `popBottom`: pop bottom node off of deque
- `pushBottom`: push node to bottom of deque
- `isEmpty`: return true if there are no nodes in deque
- `inStealableSet`: return true if this deque is to be found in a worker thread's stealable set

During execution of the algorithm, deques are in one of the following states:

- **Active**: It is the designated active deque of a given worker thread (the worker thread treats this as its local deque).
- **Suspended:** The bottom-most node that was last executed by a worker thread encountered a blocking operation, and is now waiting out the latency of the operation. This node is *not* in the deque; it will later be pushed onto the deque again by a callback when the operation completes. The deque may still contain other ready nodes that are available for worker threads to steal. TODO: make sure "ready" nodes is defined

- **Resumable:** All nodes in the deque are ready, but is not being actively worked on by a worker thread. These nodes can be stolen off the top of the deque by worker threads and then executed.
- **Muggable:** The entire deque can be mugged by a worker thread, to become the threads new active deque.

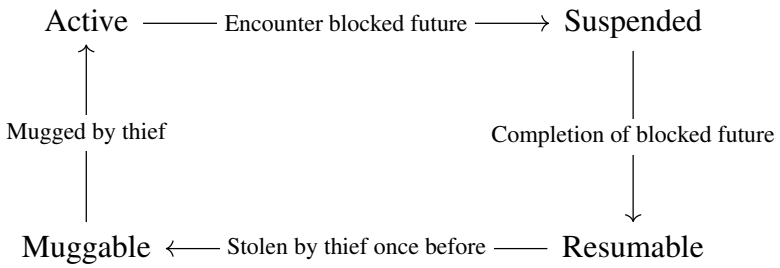

Figure 3.1: Deque state transitions

Deque transitions are displayed in figure 3.1. Deques begin their lives in the Active state, when they are first created by a worker thread. A worker thread then works off the bottom of this deque, until it encounters a blocked future node, at which point the deque becomes Suspended. Upon completion of the previously blocked future node, a callback is executed that transitions the deque into the Resumable state. At this point, any worker thread is able to steal a single node off the top of the deque. After a worker thread has stolen a node off the top, the deque transitions into the Muggable state. In this state a worker thread can steal the entire deque at once (a mugging), and become the worker thread's new active deque.

Stealable Sets

In order to keep track of stealable deques, each worker thread owns a ***stealable set***. This set contains deques that have ready nodes that are ready to be executed (i.e. available work for worker threads to perform). Like deques, stealable sets are assumed to support concurrent operation and take constant amortized time. During scheduler execution, thieves select a victim worker thread uniformly at random to steal from, and from within that victim's stealable set, uniformly at random select a victim deque.

Stealable sets support the following operations:

- `add`: add a deque to the set
- `remove`: remove a deque from the set
- `chooseRandom`: return a random deque from the set (without removing it)

3.1.4 Scheduling Loop

The main scheduling loop is shown in algorithm 1. Execution starts by setting the active deque of all worker threads to an empty deque, and pushing the root of the computation to one of the deques, after which the scheduling loop begins (line 2).

Without the extra logic to deal with futures (lines 11 - 16), ProWS-R behaves the same as classic work-stealing.

Algorithm 1 Main Scheduling Loop (w is the currently executing worker thread)

```
1:  function SCHEDULINGLOOP
2:      while computation is not done do
3:          node ← findNode()
4:          left, right ← execute(node)
5:          if left ≠ null then
6:              w.active.pushBottom(left)
7:          end if
8:          if right ≠ null then
9:              w.active.pushBottom(right)
10:         end if
11:         if node encountered blocked future f then
12:             deq ← w.active
13:             w.active ← null
14:             suspendDeque(deq)
15:             f.installCallBack(deq)
16:         end if
17:     end while
18: end function
```

Algorithm 2 Find Node (w is the currently executing worker thread)

```
19: function FINDNODE
20:     node ← null
21:     if w.active ≠ null then
22:         node ← w.active.popBottom()
23:     end if
24:     if node = null then
25:         node ← steal()
26:     end if
27:     returnnode
28: end function
```

Worker threads work off the bottom of their active deques. When a node is popped from the bottom, it is first executed and then its children (if any) are pushed to the bottom of the deque. Special care is taken for when a node that was just executed is found to have encountered a blocked future (line 11): the worker thread's active deque is immediately suspended and a callback is installed on the future that will reschedule it for execution again once it's latency-incurring operation completes. Deque suspension for the active deque (line 29) involves changing its state to Suspended, removing it from the worker thread's stealable set, and if it is not empty, adding it to the stealable set of another randomly selected worker thread. If the deque is empty, it will not be found in any stealable set, so that worker threads can not try to fruitlessly steal from it.

Algorithm 3 Deque Suspension (w is the currently executing worker thread)

```
29: function SUSPENDDEQUE(deq)
30:     deq.state ← SUSPENDED
31:     w.stealableSet.remove(deq)
32:     if !deq.isEmpty() then
33:         chooseRandomVictim().stealableSet.add(deq)
34:     end if
35: end function
```

The callback (line 36) that is installed on the blocked future (line 15) is the critical aspect for enabling latency-hiding. In order to try and keep worker threads busy with work as much as possible so that progress is being made on the computation with full hardware resource utilization, it's undesirable for worker threads to perform any operations that are not directly related to executing work nodes (contributes to scheduler overhead). As such, the callback is responsible for executing when completion of a blocked future is detected, and rescheduling the future node for execution by a worker thread. A more concrete reasoning and explanation of how this occurs is given in section 3.2.

Algorithm 4 Callback Procedure (called upon completion of the blocked future f)

```
36: function CALLBACK(suspendedDeq)
37:     if suspendedDeq.state ≠ SUSPENDED then
38:         return
39:     end if
40:     suspendedDeq.pushBottom(f)              ▷ f is a node that can be executed
41:     suspendedDeq.state ← RESUMABLE
42:     if !suspendedDeq.inStealableSet() then
43:         chooseRandomVictim().stealableSet.add(suspendedDeq)
44:     end if
45: end function
```

Due to the way Rust futures implement their completion signaling mechanism using wakers (introduced in section 2.3.1), it is possible for multiple wakers to wake up the same future. This is unlike the futures supported by ProWS, and it is vital that ProWS-R take specific care to handle *repeatedly polled futures*. Multiple polled futures can happen, for example, when two futures are manually polled immediately after one another using the same waker, like when using the `join` function [1]. Fortunately, supporting this is trivial [3]: all that is required is a check (line 37) to see if the deque

[1] The `join` function in the Rust futures crate [2] [5] can be used to create a future that concurrently executes two or more futures (note: not in parallel). It does this by polling the two or more futures passed to it in sequential order, passing and cloning its waker every time. This means as soon as at least one of the futures is ready to make progress the `join` future is awoken and can be rescheduled for execution. These multiple futures can all trigger the waker clones that wake up the same future, hence the need for ProWS-R to support repeatedly polled futures.

[3] While trivial, early versions of the implementation incorrectly assumed that when a callback is executed, the suspended deque associated with that callback will always be Suspended. This led to

the future was suspended with is already unsuspended, and if so, the callback does not perform any actions. If, however, the deque is still suspended, the callback pushes the future back to the bottom of the deque, transitions it to Resumable, and adds it to a random worker thread's stealable set if the deque is not already in one. Doing this the callback makes the now-resumable future ready to be executed by a worker thread again.

When a worker thread cannot find work to execute in its active deque (line 24), it must become a thief and steal from elsewhere. The steal procedure is outlined in algorithm 6. First, a random victim deque from a random worker thread's stealable set is chosen (line 49). Recall that stealable deques can either have nodes stolen off the top of them, or be mugged in their entirety. Given the victim deque, if it's in the Muggable state, the thieving thread mugs the entire deque and sets it to be its new active deque. Otherwise, the thieving thread attempts to pop a node from the top of the victim deque. If after popping a node the victim deque is empty, it is removed from the victim worker thread's stealable set so that other worker threads cannot futilely attempt to steal from it, and possibly even freed if not in the Suspended state (a Suspended deque can be empty but still be awaiting a callback to push a resumable future back on to it, so should not be freed). If the victim deque is Resumable it is then marked as Muggable, and a new deque is created for the thieving worker thread if it has none (which is the case when a blocked future is encountered on line 11). If a node could not be stolen, the steal procedure is repeated.

The calls to `rebalanceStealables` on lines 56 and 75 are to balance the load of stealable deques among the worker thread stealable sets. This is done so that the chance of selecting a stealable deque given a victim worker thread stays uniform. This is performed when a deque has been removed from a worker thread v's stealable set - it randomly chooses another victim v' and if $v = v'$ nothing is done, otherwise a stealable deque is moved from v' to v if v' has one.

3.1.5 Performance Bounds

As ProWS-R is effectively equivalent to ProWS in terms of complexity (ProWS-R is actually a slightly stripped down version of ProWS, with additional simple constant time operations to support Rust futures), it inherits the performance bounds of ProWS [41, 40]. Singer et. al show the execution time bound of ProWS is $O(T_1/P + T_\infty \lg P)$. This means the bound is *independent* of the number of latency-incurring operations in the computation, thus hiding latency. Compared to the classic work stealing bound of $O(T_1/P + T_\infty)$ which provides linear speedup when $T_1/T_\infty = \Omega(P)$, ProWS, and by extension ProWS-R, provide linear speedup when $T_1/T_\infty = \Omega(P \lg P)$.

Briefly, the analysis of ProWS achieves a bound independent of the number of latency-incurring operations by exploiting the fact that stealable deques must be stolen from once while in the Resumable state before transitioning to the Muggable state. By stealing once before mugging the entire deque, this ensures that for each mugging

all sorts of extremely subtle and difficult to diagnose issues where scheduler state eventually became corrupted, since nodes would be wrongly pushed to deques more than once, possibly leading to the program never terminating or other consequent problems.

Algorithm 5 Steal Procedure (w is the currently executing worker thread)

```
46: function STEAL
47:     while true do
48:         victim ← chooseRandomVictim()
49:         victimDeque ← victim.stealableSet.chooseRandom()
50:         if victimDeque.state = MUGGABLE then
51:             return setToActive(victim, victimDeque)
52:         end if
53:         node ← victimDeque.popTop()
54:         if victimDeque.isEmpty() then
55:             victim.stealableSet.remove(victimDeque)
56:             rebalanceStealables(victim)
57:             if victimDeque.state ≠ SUSPENDED then
58:                 freeDeque(victimDeque)
59:             end if
60:         else if victimDeque.state = RESUMABLE then
61:             victimDeque.state ← MUGGABLE
62:         end if
63:         if node! = null then
64:             if w.active = null then
65:                 w.active ← createNewDeque()
66:             end if
67:             return node
68:         end if
69:     end while
70: end function
```

Algorithm 6 Set to Active Deque Procedure (w is the currently executing worker thread)

```
71: function SETTOACTIVE(victim, victimDeque)
72:     victim.stealableSet.remove(victimDeque)
73:     w.stealableSet.add(victimDeque)
74:     victimDeque.state = ACTIVE
75:     rebalanceStealables(victim)
76:     if w.active.isEmpty() then
77:         freeDeque(w.active)
78:     end if
79:     w.active ← victimDeque
80:     return w.active.popBottom()
81: end function
```

there is a corresponding steal to amortize against, allowing the number of steals to be bounded. Since a work-stealing scheduler is either working or stealing, the total running time is $(T_1 + X)/P$, where X bounds the number of steal attempts. Armed with a bound on the number of steals, the final bound on execution time can be found.

3.2 Required Runtime Support for Latency-Hiding

The ProWS-R algorithm on its own is not enough to enable latency-hiding [4]. To truly support latency-hiding, additional runtime special considerations must be accounted for to support the core scheduling algorithm. Scheduling futures is one thing, but actually hiding the latency in an efficient manner is another. Essentially, the runtime support needs to answer the question: how can latency-incurring operations be performed asynchronously, while worker threads can still make progress on the primary computation?

3.2.1 The I/O Thread

The crux of the problem is that given a fixed number of P worker threads, it is undesirable for any of the P worker threads to be doing anything except for executing nodes. Anything that a worker thread does outside of this only contributes to scheduler overhead. To avoid placing the burden of processing latency-incurring operations on a worker thread, the ProWS-R runtime uses an additional thread, named the I/O thread, dedicated solely to this task [5]. This relieves the worker threads of having to sacrifice time that could otherwise have been spent executing work.

Naturally, at first glance this may seem to bring little benefit, as introducing an additional thread simply means that now hardware resource usage needs to be split among $P+1$ threads [6]. Although this is true, the runtime can take advantage of the fact that the I/O thread can simply be put to sleep whenever its services are not required (i.e. if there are no latency incurring operations to process). When the I/O thread is put to sleep, the underlying operating system thread scheduler can dedicate the entirety of the available hardware resources to the P worker threads [7], with the I/O thread not taking up any processor cycles.

What remains is to see how the worker threads and I/O thread interact to process latency-incurring operations. The following functionality is required:

[4] This section is based off the work by Singer et. al on Cilk-L [40], a latency-hiding extension of Cilk that uses the ProWS algorithm, with considerations on how to integrate with the mechanisms involved with Rust futures.

[5] The I/O thread capability is provided by the async-io crate [8], which abstracts over varying operating system event queues

[6] Classic work-stealing runtimes create P worker threads for P physical processor cores, to maximize hardware usage efficiency [12].

[7] This is a slight oversimplification: in principle the operating system thread scheduler can dedicate all hardware resources to the P threads, but of course in reality on a modern computing platform, other programs may be running on the same machine and/or the underlying thread scheduler may not be aware of the nature of the work-stealing worker threads. Fortunately, it can be shown that work-stealing is optimal to a constant factor even in the face of such an adversarial thread scheduler [12].

1. When a worker thread encounters a blocked future, it must somehow register this with the I/O thread and delegate responsibility of dealing with the blocked future, so that the worker thread can return to executing work as soon as possible.
2. The I/O thread, upon registration of a blocked future by a worker thread, must monitor the blocked future to detect when it completes. Once complete, the I/O thread must perform the callback in algorithm 4 to make the future available for a worker thread to resume again.

One strategy to fulfill these requirements would be for the I/O thread to repeatedly poll to see if the file descriptors that futures are blocked on have become ready. This, however, is not ideal as it would necessitate the I/O thread to take up processor resources performing this repetitive polling, even when nothing is ready. An additional concern would be how often to perform the polling: too often and processor usage would be excessive; not often enough and resumable futures might not be made available quickly enough.

3.2.2 Event Queues

To avoid these issues, this functionality is instead achieved by relying on the underlying operating system event queue: epoll on Linux, kqueue on BSD systems, and IOCP on Windows. The I/O thread has an instance of such an event queue. When a worker thread encounters a blocked future, it registers the desired file descriptor that the future is blocked on with the event queue of the I/O thread (note that this is done by the worker thread, not the I/O thread itself). Once it has done this, the worker thread can proceed with executing other work. Since registration of the file descriptor is done by the worker thread, this means the I/O thread need not wake up. This achieves part 1.

The I/O thread waits on events provided to it by the event queue: if there are no events to process, the I/O thread goes to sleep. When any events are ready, the event queue wakes the I/O thread, at which point the I/O thread can then execute the callback (algorithm 4) to make the previously blocked future (registered by a worker thread in part 1) available for a worker thread to resume again. More concretely, when the underlying resource a future was blocked on becomes ready, the I/O thread triggers the corresponding waker [8] which then executes the callback. This achieves part 2.

By relying on the underlying operating system event queue, the I/O thread only ever uses processor cycles whenever a blocked future becomes resumable, and needs its corresponding callback executed. At all other times it is asleep, and the available processing resources can instead be fully utilized by the worker threads to execute work.

[8] As described in section 2.3.1, the waker mechanism is used for signaling if futures are ready to make progress. When the I/O thread, awoken by the event queue, detects that a future is ready to make progress, its waker will be triggered (the waker will then execute the callback in algorithm 4). Typically, Rust futures simply wrap other futures (that are then compiled into one large future, represented by a state machine), so the responsibility of triggering a waker to signal that a given future is ready to make progress can simply be delegated to the nested future (the outer future will block on the inner future, so if the inner future can make progress then so can the outer future). A leaf future (a future that contains no nested futures), however, has no nested future to pass this responsibility down to: instead, it registers the resource it is blocked on with the I/O thread event queue (part 1).

In between the time a worker thread encounters a blocked future and the future becomes resumable, it performs useful work, thus hiding the latency-incurring operation of the blocked future.

Chapter 4

Implementation: Time is an Illusion

This chapter describes the technical details and experience of the prototype implementation of ProWS-R and its corresponding runtime. While the conceptual concepts of ProWS-R described in chapter 3 suffice for a theoretical latency-hiding work-stealing scheduler, naturally in practice additional considerations need to be taken. Presented only with the high-level scheduler algorithms (algorithms 1, 4, and 6), questions regarding an efficient and idiomatic real-world implementation still remain. Concretely:

- How are work nodes represented, in a library-level work-stealing implementation (as opposed to a compiler based implementation such as in Cilk)?
- How are work nodes that represent latency-incurring operations represented, particularly in regards to facilitating the use of the Rust futures language feature?
- How is shared scheduler state managed and updated concurrently (thread safety), without detrimental performance impacts?
- Where are work nodes and deques stored in memory? How is memory safety ensured (e.g. blocked futures must not have their state and data freed before resumption)?
- How does the scheduler implementation hook into the underlying operating system event queue?

Presented in this chapter is a detailed description of how the proof of concept implementation done as part of this project answers the above questions. Section 4.1 provides an architectural overview and section 4.2 explains how work nodes are represented. Section 4.3 outlines the specific contributions of the implementation. The implementation experience and challenges are then described in sections 4.4 and 4.5.

4.1 Rayon-LH Architecture Overview

As the focus of this project is to present a proof of concept implementation of latency-hiding work-stealing that integrates with the Rust language and its support for asynchronous programming, the implementation is a fork (henceforth referred to as Rayon-

LH) [45] of the widely used Rayon library [10, 2, 44]. Rayon is a library-level implementation of classic work-stealing for Rust programs. Since Rayon is a classic work-stealing implementation, it does not account for latency-incurring operations: users are explicitly warned not to schedule I/O or other latency incurring operations for risk of causing worker threads to block and performance suffering [6]. The desire to mix both compute heavy and I/O operations in a more general purpose Rayon style interface is a common refrain by users [3, 39]. The goal of the Rayon-LH implementation is to provide the ability for users to easily parallelize their programs that involve not just compute bound operations, but operations that incur latency as well.

Building on top of Rayon allows Rayon-LH to take advantage of the existing infrastructure for classic work-stealing, and focus on extending the library with latency-hiding capabilities. Here a brief overview of the entire architecture of the implementation is presented, with details on the differences with Rayon.

It perhaps is best to first take a step back and understand the concrete data structures involved in Rayon-LH. Doing so will provide a general understanding of the layout of the implementation, and the following sections can then describe the specific interactions involved between the various components as they execute the ProWS-R scheduling algorithm. The core components of Rayon-LH are broken down into the following: the ***Registry*** (central thread pool), ***Deques*** (concurrent work-stealing deques), ***StealableSets*** (stealable sets containing stealable deques), the ***Stealables*** construct (abstracts over StealableSets and manages scheduler shared state), ***WorkerThreads*** (worker threads in thread pool), and ***Jobs*** (represent work nodes). Additional minor implementation details deemed not critical to understanding the implementation are omitted.

TODO: put in a diagram of the architecture

The Registry

The life of the scheduler begins with the Registry struct [1]: a single global Registry instance [2] is created for a given program. The Registry can essentially be thought of as a thread pool consisting of WorkerThread structs. It is responsible for creating the WorkerThreads that will then execute the primary scheduling loop in algorithm 1.

Importantly, it also stores the `deque_bench` and `injector` . The `deque_bench` is a concurrent sharded hash map (provided by the DashMap crate [27]) that stores Deques that are not currently the active deque of a WorkerThread (the reasoning for this will be explained in subsequent sections). The `injector` is a concurrent FIFO queue used to inject Jobs from outside of the threadpool (the primary purpose of this queue is to inject the root work node of a computation into the thread pool from the main thread of the program).

[1] Structs in Rust are similar to structures in C (a struct consists of only data fields). Rust adds the ability to define functions that accept a given struct as a special parameter and use them with traditional OOP-like method syntax, making them, at least on the surface, appear similar to traditional objects found in languages with OOP features.

[2] Rayon and Rayon-LH actually support creating multiple Registry instances to represent multiple thread pools, but this feature does not affect the implementation design much so will not be discussed.

After initializing *P* WorkerThreads for *P logical* cores (this will further be discussed in the evaluation in chapter 5) that execute the main scheduling loop, the Registry waits for WorkerThreads to indicate computation is fully complete (no more Jobs can be found). The Registry is created on the program main thread, and will block and go to sleep waiting for this completion indication from the WorkerThreads [3].

Deques

Deques are work-stealing double-ended queues that support the concurrent operations described in section 3.1.3. A Deque contains nodes in the computational DAG representing work to be executed; in Rayon-LH nodes are implemented as ***Jobs*** (discussed in section 4.2). Each Deque is assigned a unique index, so that it may be uniquely referenced throughout the scheduler. The primary purpose of this index is to provide quick constant time retrieval (as keys in a hash map) of deques from the Registry `deque_bench` and the Stealables mapping (discussed shortly). References (essentially pointers in Rust) cannot be used, since this would lead to pointer invalidation (dangling pointers) as the `deque_bench` or Stealables mapping is updated (not to mention likely incredible pain dealing with lifetimes from the Rust borrow checker, that tries to prevent such programming practices). Apart from the unique index, the Deque struct is just a simple wrapper around the work-stealing deque implementation provided by the Crossbeam crate [9].

A key thing to note though, is that the Crossbeam deque implementation actually provides two handles to the underlying deque: a `Worker` and a `Stealer`. A `Worker` handle allows pushing and popping to the bottom of the deque, and only one such handle may be created (since it is **not** thread-safe). A `Stealer` handle allows only stealing from the top of the deque, but multiple may be created (and **is** thread-safe). This has implications for Rayon-LH, as this means multiple `Stealers` to the same deque can be created and used by different worker threads in a thread-safe manner throughout the program, but only a single `Worker` (giving access to the bottom of a deque) may be in use by a single thread at a time (note that it can still be *sent* to other threads, just may not be used concurrently).

As such, a Deque struct is more specifically a wrapper around a `Worker`: only a single thread may access and use a Deque struct at a time [4]. Rayon-LH takes very special care to achieve this: by relying on Rust's ownership system (destructive moves) and never creating references to a Deque or anything within it.

[3] Rayon uses a thread sleep/notification mechanism based on atomic latches and counters, an overview can be found in [11].

[4] Those more experienced with Rust may point out that the `Send` and `Sync` traits should address these issues: unfortunately since `Workers` are `Send` but not `Sync`, there are many places that the compiler will not allow Deque structs to be used, as it falsely presumes Deques will want to be used concurrently. Since Rayon-LH takes care to only ever use Deques in a non-concurrent manner (by only passing around ownership, and never creating references), an `unsafe impl Sync` is provided for Deques to resolve this.

StealableSets

StealableSets, as the name suggests, are used to represent stealable sets in ProWS-R, and support the operations described in section 3.1.3. Instead of storing Deques, however, StealableSets only store the unique IDs of Deques (Deques are *stored* in the Registry `deque_bench`; StealableSets only need to keep track of which deques can be *found* in a given deque set). Although it may seem trivial to implement a set of stealable deques by simply storing them in a set, the `chooseRandom` operation unfortunately precludes such a simple approach: sets typically do not support constant time retrieval of a **random** item [5]. To solve this, Rayon-LH uses a resizable array containing Deque IDs and a hash map of Deque IDs to their index in the array. When a random deque ID is requested, a random Deque ID in the array is selected and returned (constant time operation). The hash map is required for a constant time `remove` operation: the index of the desired deque ID to be removed from the set is looked up in the hash map, the Deque ID in the array at that index overwritten with the Deque ID at the end of the array, the array truncated by one, and the hash map indices updated/removed. All operations are guarded by a lock to support concurrent operation. TODO: should I remove all this fluff on implementation detail

The Stealables Struct

The Stealables struct is used to abstract over the P stealable sets, one for each WorkerThread struct. A single Stealables struct is created during Registry creation, and is kept alive in the system by having each WorkerThread hold an atomically reference-counted smart-pointer to it. The primary purpose of the Stealables struct is the Stealables mapping (the `deque_stealers` field): this is a concurrent sharded hash map of Deque IDs to the Deque state, the worker thread whose stealable set they can be found in (if any), and the `Stealer` of the corresponding Deque. This information cannot simply be stored in the Registry `deque_bench` as it does not include Deques in the Active state (why not is explained shortly). This Stealables mapping contains absolutely crucial shared scheduler state, and is used by WorkerThreads to coordinate and execute the ProWS-R algorithm [6]. Abstracting over the stealable sets is necessary, since modifications to stealable sets naturally modify scheduler state, so the Stealables mapping must always be updated to reflect these changes.

[5] Many implementations indeed support iteration over the items in a set, and the reader may wonder why Rayon-LH can not simply iterate a random number of times to retrieve a randomly selected item. Unfortunately, since stealable sets may potentially contain a large number of deques (every time a suspended future is countered its deque is added to a stealable set), this would require a $O(n)$ iteration to get a random item from the set.

[6] The Stealables mapping was a massive source of headaches stemming from concurrency bugs that eventually corrupted scheduler state. This is because although the concurrent hash map itself is thread-safe, a sequence of multiple operations is not guaranteed to be completed as an atomic unit. An atomic sequence of operations can only be performed while a reference into the hash map is held. This means if multiple operations to the Stealables mapping in one atomic unit is required (which ProWS-R frequently does), very special care must be taken to not drop the reference into the mapping while the operations are being performed. Figuring out which sequence of operations must be performed as an atomic unit was also not trivial.

WorkerThreads

WorkerThreads are perhaps the life and soul of Rayon-LH: they are the components that execute the main ProWS-R scheduling loop and drive scheduler progress. Each of the *P* WorkerThreads contains its respective active Deque, an atomically reference-counted pointer to the Registry struct, and an atomically reference-counted pointer to the Stealables struct. The *P* WorkerThreads run the scheduling loop, performing the steal operation (algorithm 6) when necessary (with the addition of checking the global Registry `injector` queue if cannot find work in both its local deque or by stealing), interacting with the various components described above.

A key thing to note is that WorkerThreads take ownership of their active Deques, and as such Deques in the Active state are not found in the Registry `deque_bench`. Recall that Deques wrap a `Worker` instance that must not be used concurrently: as mentioned, this is achieved by relying on Rust's ownership system and never creating references to Deques. By not creating references and only passing around ownership of Deques, the Rust compiler can ensure that only a single thread has access to a given Deque at a time [7]. Deques are either owned by a single WorkerThread (if they are in the Active state), where they can only be accessed by that WorkerThread, or are stored in the `deque_bench`. Although all threads in the system have access to the `deque_bench`, the only time a Deque in the `deque_bench` is used in a manner that would not be thread-safe is when the I/O thread pushes a resumable future Job back on the bottom of its suspended Deque when executing the callback in algorithm 4. However, this is only performed by the *single* I/O thread, and hence does not pose a danger.

4.2 Jobs: Representing Work

While conceptually a work node in the computational DAG is simple to understand, how is this actually implemented in practice? Cilk and its derivatives take an approach of using a combination of a compiler and runtime library to essentially manually create closures [8] (since the C language itself provides no such feature) that represent work nodes in the computational DAG stored on the respective stacks of worker threads where they were created, and *pointers* to these closures are stored in the deques that are subsequently popped and stolen to be dereferenced and the corresponding closure executed (using calls to `setjmp` and `longjmp` inserted by the Cilk compiler) [9] [22].

Rust has the fortunate advantage of supporting closures as a built-in language feature. This means in Rayon-LH, the role that the Cilk compiler plays in creating closures can more or less be performed by the Rust compiler itself, lending the language to a library-level work-stealing implementation. While in Cilk closures represent work

[7] An `unsafe impl Sync` is still required since Deques are used in situations where it would be *possible* for them to be used concurrently, and the compiler is not aware of the intentions of the Rayon-LH implementation. Consequently this manual opting-out of the compiler protections must be used.

[8] In this context: functions with captured references to their enclosing stack frames. In Cilk this is done by creating Cilk "activation frame" structures. In Rust, closures are basically compiler generated structs that contain references or copies to the closure's environment, and a method that executes the closure's function using this captured environment.

[9] Please note this brief explanation is a massive oversimplification of the Cilk implementation.

nodes in the DAG, in Rayon-LH work nodes are represented by the `Job` trait [10] (shown in figure 4.1): anything that implements the `Job` trait (i.e. a "Job") can be used as a node in a Deque.

```
trait Job {
    unsafe fn execute(this: *const Self);
}
```

Figure 4.1: `Job` trait (simplified)

Similar to how in Cilk deques store pointers to closures, in Rayon Deques do not directly store types that implement the `Job` trait, but instead "fat pointers": dynamically dispatched Job types. This allows arbitrary concrete types that implement the `Job` trait (note the `this` parameter in `execute` is a pointer type, rather than a concrete type) to be stored in Deques as a single homogeneous fat pointer Job. This fat pointer type is the `JobRef` struct (basically two pointers), shown in figure 4.2, and is what is stored in Deques. The `pointer` field is a pointer to the captured environment and other metadata, and the `execute_fn_pointer` field is a function pointer to the concrete implementation of the `execute` function of the concrete type that implements `Job` [11]. WorkerThreads, when executing the scheduling loop, push, pop, and steal these `JobRef` objects, and execute the work node they represent by calling `execute_fn_pointer` and passing in the `pointer` field.

```
#[derive(Copy, Clone)]
pub struct JobRef {
    pointer: *const (),
    execute_fn_pointer: unsafe fn(*const ()),
}
```

Figure 4.2: `JobRef` type (simplified)

StackJobs

The primary Job in Rayon is represented by the generic `StackJob` type, shown in figure 4.3: a thin wrapper around a Rust closure, a location in memory (the stack of the WorkerThread) to deposit the final return value of the closure, and a corresponding latch (a synchronization primitive). `StackJobs`, as the name implies, are stored on the stack of the WorkerThread that created them. WorkerThreads create `StackJobs` and push the corresponding `JobRefs` to their active Deques (unless the `StackJob` was created on the program main thread, for example the root computation node, in which case the `JobRef` is pushed on the global Registry `injector` queue). The `execute_fn_pointer` for `StackJobs` basically just executes the wrapped closure and trips the latch.

Care must be taken by the WorkerThread to not clobber the stack (since the stack stores the closures environment) while a `StackJob` has not yet been executed: this

[10] Traits in Rust are similar to interfaces or type classes in other languages.

[11] Readers experienced with Rust may recognize this as a hand-rolled trait object. A hand-rolled version using raw pointers is used to avoid lifetime quarrels with the Rust borrow checker (care is taken to ensure safety).

```
pub struct StackJob<L, F, R>
where
    L: Latch + Sync,
    F: FnOnce(bool) -> R + Send,
    R: Send,
{
    latch: L,
    func: F,
    result: JobResult<R>,
}
```

Figure 4.3: `StackJob` type (simplified)

is achieved by the WorkerThread performing the scheduling loop looking for Jobs to execute for as long as the `StackJob` latch has not yet been tripped, and whatever WorkerThread that finally executes the `StackJob` tripping the latch to indicate to the original WorkerThread that it is now safe to pop the stack frame the `StackJob` lives on.

As `StackJobs` are generic (parametric polymorphism resolved at compile-time), a unique `StackJob` type is created for every unique function pointer or compiler-generated closure it wraps. This is where the dynamically dispatched nature of the `JobRef` struct truly shines: an infinite number of concrete `StackJob` or any other Job type that can provide an `execute` function through the `Job` trait can be represented using a single uniform `JobRef` type.

FutureJobs

The `StackJob` type is intended for regular closures: functions that can not be suspended/resumed and do not incur latency. To support asynchronous operations that represent future nodes in the computational DAG, Rayon-LH uses the generic `FutureJob` type shown in figure 4.4. A `FutureJob` is similar to a `StackJob`, but wraps a Rust Future (described in section 2.3.1) instead of a regular Rust closure. Note that a `FutureJob` is not a Rust future itself, it merely wraps such a future.

```
pub struct FutureJob<L, F>
where
    L: Latch + Sync,
    F: Future,
{
    latch: L,
    future: F,
    result: JobResult<F::Output>,
    waker: FutureJobWaker,
}
```

Figure 4.4: `FutureJob` type (simplified)

In order to provide the guarantee that a Rust future stay pinned in memory once it

has first been polled, the API (figure 4.5) to create, spawn, and await a `FutureJob` relies on the Rust `Pin` type, as described in section 2.3.1. In short, the Rust type system statically guarantees that once a `FutureJob` is spawned, it is not moved from its original memory location.

```
let future_job = rayon::FutureJob::new(network_request());
pin_mut!(future_job); // pin future_job on stack
let future_job_handle = future_job.spawn();
// ... execute some code in parallel while
// ... future_job hides its latency
let result = future_job_handle.await_future_job();
// future_job guaranteed to not have moved in memory
// by the time future_job is awaited
```

Figure 4.5: `FutureJob` API, demonstrating pinning on the stack

The primary difference to a `StackJob` is the addition of a `FutureJobWaker`, and what happens in `execute_fn_pointer`. When executing `execute_fn_pointer`, a `FutureJob` creates a `FutureJobWaker` and polls its wrapped future with it. If the future returns `Poll::Ready` indicating that the future is complete, the final return value is simply deposited and the latch is tripped, just like a `StackJob`. Otherwise, if the future returns `Poll::Pending` indicating the future is blocked, lines 11 - 16 of the scheduling loop are performed: the active Deque of the executing WorkerThread is suspended, and the resource the future is blocked on is registered with the I/O thread event queue (section 3.2.2). The previously active Deque is put back in the Registry `deque_bench` so that it may have somewhere to live in memory.

At some point in the future when the I/O thread event queue detects the resource is ready to be used, the I/O thread is awoken and triggers the `FutureJobWaker`. The `FutureJobWaker` executes [12] the callback procedure of algorithm 4: the `JobRef` representing the previously blocked `FutureJob` is pushed back on the bottom of the suspended Deque, the Deque is transitioned to the Resumable state, and potentially added to a StealableSet for a WorkerThread to execute again. This harmonization of `FutureJobs`, the I/O thread, and the `FutureJobWaker` are what provide the latency-hiding capabilities of ProWS-R.

4.3 Differences between Rayon and Rayon-LH

Although Rayon-LH builds upon much of the existing Rayon infrastructure, much of it needed to be heavily modified and new infrastructure added to support ProWS-R. While the overarching approach of representing work nodes as `JobRefs` stored in Deques that are worked on by WorkerThreads is the same, every single component outlined in section 4.1 had to be overhauled or added (the Stealables and StealableSets components are new additions).

[12] Note it is the single I/O thread that is executing the callback procedure, not a worker thread. This takes care of the issue that the `Worker` handle to a deque is not safe for concurrent use, discussed in section 4.1.

A key contribution of Rayon-LH is also the `FutureJob`: this component is what provides the ability to integrate Rust futures with ProWS-R in a seamless manner. The `FutureJob` neatly takes advantage of the Rust waker machinery to efficiently provide latency-hiding work-stealing capabilities. The original Rayon scheduling algorithm and steal procedures were completely gutted and replaced by the respective ProWS-R procedures. At the same time, ProWS-R provides almost a sort of "backwards compatibility" with the tried and true classic work-stealing algorithm, in the sense that if latency-incurring operations (i.e. `FutureJobs`) are not used, the scheduling algorithm is the exact same and also provides the exact same performance (discussed in chapter 5).

Arguably, a weakness of Rayon-LH is that it requires far more shared state to perform its scheduling capabilities than does Rayon. All WorkerThreads make frequent use of the Stealables mapping (section 4.1), which means extreme care needs to be taken to ensure concurrency correctness. Rayon, while it does also have its share of hair-raising concurrency traps for things like thread sleeping and notification (that Rayon-LH also inherits), has no need for WorkerThreads to interact with a shared Stealables-like component.

4.4 Pitfalls

A complete memoir of the migraine-inducing false assumptions and traps encountered during the implementation of Rayon-LH could likely fill an entire tome (with possibly yet even more undiscovered). Here is a short non-exhaustive list:

- The `FutureJobWaker` must ensure that it begins execution of the callback procedure in algorithm 4 only *after* the deque suspension process on line 29 is fully complete. Without a synchronization primitive like a latch, it could be the case that the callback executes too soon and insidiously corrupts state.
- The `FutureJobWaker` must explicitly notify a WorkerThread that work is available again after executing the callback procedure where it pushes the previously suspended `FutureJob` back on the Deque. Otherwise it is possible all WorkerThreads are asleep, yet there is work to be executed, and the program never terminates.
- Unlike in the pseudocode of algorithm 6, in practice the steal procedure retries only a fixed number of times before the WorkerThread goes to sleep (to avoid futilely attempting to steal if there is no work available). Consequently, particularly if running on a machine with a large number of processors, it could be the case that all awake WorkerThreads fail to randomly select the one victim StealableSet (line 48) that has work available. Since no WorkerThread selects this StealableSet, no WorkerThread finds any work, and all WorkerThreads go to sleep. Again, there is work to be executed but no WorkerThreads awake to execute it, and the program never terminates. Rayon-LH deals with this by adding a final brute-force linear scan of StealableSets, to search for a stealable Deque if all previous random steal attempts failed.

- The general concept of fallible operations that must be retried. Due to the nature of WorkerThreads running in parallel, it's clearly possible that a WorkerThread beats another WorkerThread to performing an operation. What is less clear is how to deal with this in ProWS-R: a common approach taken in Rayon-LH is to allow for operations to fail, but retry a fixed number [13] of times.
  - An example of this is in the steal procedure: two WorkerThreads may randomly select the same victim Deque in the Muggable state, but only one will succeed in mugging it. The other WorkerThread must realize its desired deque has disappeared before it got to it, take care not to mistakenly corrupt state, and finally retry the steal procedure.

4.5 Implementation Challenges: Concurrent Programming is Hard

Undoubtedly, the experience of implementing Rayon-LH was quite challenging. The difficulties faced in this project largely stem from the inherent issue that concurrent programming is simply hard to wrap ones mind around. The intractable number of thread interleavings and possible flows of execution make for a rather miserable programming experience, if one does not take proactive caution and combative measures. Even trickier phenomena like atomic operations and memory orderings only add to the pain.

A stark consequence of the non-determinism of concurrent programming is that debugging becomes even more complicated. Many heisengbugs [14] were encountered in the process of the implementation, and other bugs only showed their face after many thousands of program executions or when running into pathological input cases. To aid buggy-implementation diagnosis, heavy use of the logging system that Rayon-LH inherits from Rayon and stress tests in the form of repeatedly running the benchmarks in chapter 5 was made. While far from a panacea to the undying wish of the hardware to spite the implementation running on it, the logger output was instrumental to piecing together a narrative of why or why not the scheduler was performing certain operations and not others. Much like a detective investigating forensic evidence at a crime scene, the interleaved lines of thread logger output provided many clues when debugging Rayon-LH. Unfortunately, concurrent programming is far from "elementary, my dear Watson".

[13] In the current implementation these limits are arbitrarily set, likely a more rigorous analysis can be performed to find a more suitable number.

[14] A bug that disappears after trying to study it, particularly by running it in a debugger.

Chapter 5

Evaluation: To Superlinear and Beyond (Not Really... Just Superlinear)

This chapter evaluates ProWS-R and Rayon-LH on the following: how effective the latency-hiding capabilities are when latency is injected in section 5.2, the impact of varying the level of compute vs I/O bound workloads in section 5.3, and the scheduler overhead compared to classical work-stealing 5.4. The experimental setup is described in section 5.1 and the limitations of the implementation are discussed in section 5.5. All benchmark results and programs can be found in [7].

5.1 Experimental Setup

TODO: talk about averaging 10 runs too

5.2 Latency-Hiding Efficiency

The biggest question the evaluation seeks to answer is, how efficient are the latency-hiding capabilities? In other words, given a computational DAG, what is the impact on program execution time when nodes in the DAG incur latency. To answer this, the implementation is evaluated on a benchmark that runs a distributed map and reduce example with varying simulated network latencies, and the speedups measured. The same benchmark is used in [32] and [40], and is used here with the same parameters as both to allow for comparison.

What we want to see in a benchmark that evaluates latency-hiding efficiency for ProWS-R and Rayon-LH, is how close the speedups given by the scheduler are to an "ideal" latency-hiding scheduler. Like in [40], the ***Ideal*** scheduler is defined as the scheduler that never incurs latency on an operation. This is a bit of an unfair comparison for the Rayon-LH implementation, as this Ideal scheduler is no longer latency-hiding as much as it is latency-eliminating (*all* latency, even on the critical path, is removed), while in

reality a truly Ideal latency-hiding scheduler still suffers from the latency on the critical path, but manages to hide all other latency perfectly. But as work-stealing is intended for workloads that are work (T_1) dominated as opposed to span (T_∞) dominated, this detail does not have much of an impact on the benchmark results.

The distributed map and reduce program is a classic divide-and-conquer operation that maps a function $f(x)$ over a large set of n values, and reduces the resulting values with an associative binary operation $g(x,y)$. An identity function *id* is used for cases where a reduction with only one operand must be performed. To simulate a real-world use case, each of the n values is assumed to be stored on a remote server and must be requested, which incurs latency.

In the benchmark implementation (called MapReduceFib), $f(x)$ is simply the naive recursive parallel (with a serial base case) Fibonacci algorithm and $g(x,y)$ sums the results modulo a constant. The following parameters were used: there are $n = 5000$ "remote" connections, the n values "retrieved" over the network is the number 30 (to compute the 30th Fibonacci number), the serial base case is 25, and the modulo divisor is 1,000,000,000. In the MapReduceFib benchmark, the ProWS-R latency-hiding scheduler uses futures that suspend for a given duration to hide latencies whereas the classic scheduler simply uses a blocking sleep call (baseline is the classic scheduler with a single worker thread). The Ideal scheduler simply skips any simulated latency and immediately begins computation, completely eliminating all latency in the DAG.

The MapReduceFib benchmark speedup results of ProWS-R and the classic scheduler, compared to the baseline, are shown in figure 5.1. Simulated latencies of 1, 50, and 100 milliseconds were used. As can be seen, ProWS-R indeed manages to hide latency and achieve speedups very close to Ideal. Naturally, the benefit of latency-hiding when the latency is very short (1 millisecond) is minimal compared to when the latency is significant (50, 100 milliseconds). With a latency of 1 millisecond, the total amount of latency in the computation is not considerable when compared to the total amount of computational work T_1. Consequently the difference in speedups are not as drastic. When the latencies are increased to 50 and 100 milliseconds, the benefits of latency-hiding truly start to show. ProWS-R exhibits superlinear [1] speedups: reaching up to 297× and 517× for 50 and 100 milliseconds, respectively. This is compared to 35× for both latencies using the classic scheduler.

ProWS-R displaying larger speedups as latency values increase confirms that the scheduler is indeed effective at hiding latency: while the classic scheduler must incur the latency penalty on each latency-incurring edge, ProWS-R needs only incur such a cost once, for the longest latency-incurring on the critical path. Thus, the more latency in a DAG, the more the classic scheduler suffers, while ProWS-R is unaffected (except for the cost of the critical path of course).

Another interesting thing to note is the effect of hyperthreading [2]. When the number of

[1] Superlinear speedup is defined as a speedup *greater* than P when running on P processors [38]. This appears to contradict the work law defined in section 2.2.3, but is explained by the fact that the classic scheduler incurs *all* latency costs in T_1, while the latency-hiding scheduler incurs only the latency cost in the critical path T_∞.

[2] Hyperthreading is Intel's proprietary implementation of hardware-level threading

worker threads starts to exceed the number of physical cores, two worker threads may be running on the same physical core, but in separate hyperthreads. Once this boundary is crossed, speedups of the latency-hiding schedulers (both ProWS-R and Ideal) begin to degrade. This is presumably due to worker threads on the same physical core contending for the same physical processor execution units, as by hiding latency they can focus instead on computational work in the DAG. The classic scheduler with a latency of 1 millisecond suffers the same fate. With longer latency values, the classic scheduler sees a perfect linear speedup regardless of using more worker threads than physical cores, since when incurring latency the worker threads become I/O bound (primarily waiting around doing nothing), and the hyperthreads are no longer in contention. When using latency-hiding schedulers, or for that matter any work-stealing implementation running primarily compute-bound workloads, setting the number of worker threads P to be the number of physical processors appears to be the best choice to maximize performance.

5.3 Compute vs I/O Bound Workloads

This section seeks to answer the question of varying the ratio of compute vs I/O bound operations in the workload. Whereas section 5.2 is concerned about the effects of *injecting* latency by keeping T_1 constant and only varying the amount of latency T_{latency}, this section is interested in what happens when the *sum* of total work and latency $T_1 + T_{\text{latency}}$ is held constant but the individual parameters adjusted. In other words, how does ProWS-R perform under workloads of varying natures?

This parameter sweep benchmark runs the naive recursive parallel Fibonacci computation with a serial base case of zero, with the number of worker threads set to the number of physical cores. When the computation hits the Fibonacci base case (the leaf nodes in the computation DAG), it incurs latency by either performing a blocking sleep to simulate a compute bound operation (a compute bound operation *must* take up processor cycles, there is no hiding possible) or uses a future that suspends for the equivalent duration (but this latency can be hidden). The percentage of I/O bound nodes in the workload dictate whether the blocking sleep call or future will be used.

A plot showing the parameter sweep benchmark results is shown in figure 5.2. As expected, the greatest speedups come from when the workload consists of a high percentage of I/O bound nodes. Speedup is also increased when longer latency values are used, but not to the extent that the I/O bound node percentage does. Essentially, when there is an abundance of latency-hiding operations, the classic scheduler suffers tremendously as it must bears the cost of each individual latency penalty, while ProWS-R can efficiently hide this.

From this it is clear that the workloads that will benefit most from latency-hiding are ones with a high percentage of latency-incurring operations, and compute-bound workloads benefit less. Fortunately, in all cases, even when the workload is purely compute-bound, ProWS-R performs better or equal to classic work-stealing (a benchmark dedicated to compute-bound comparison is discussed in section 5.4).

5.4 Rayon-LH Scheduler Overhead

While latency-hiding capabilities are great, it would be less than ideal if this came at the cost of sacrificing performance on regular compute-bound workloads.

5.5 Insights and Limitations

TODO: move concurrency difficulties here, talk about pinning, stack overflow, tied to async-io

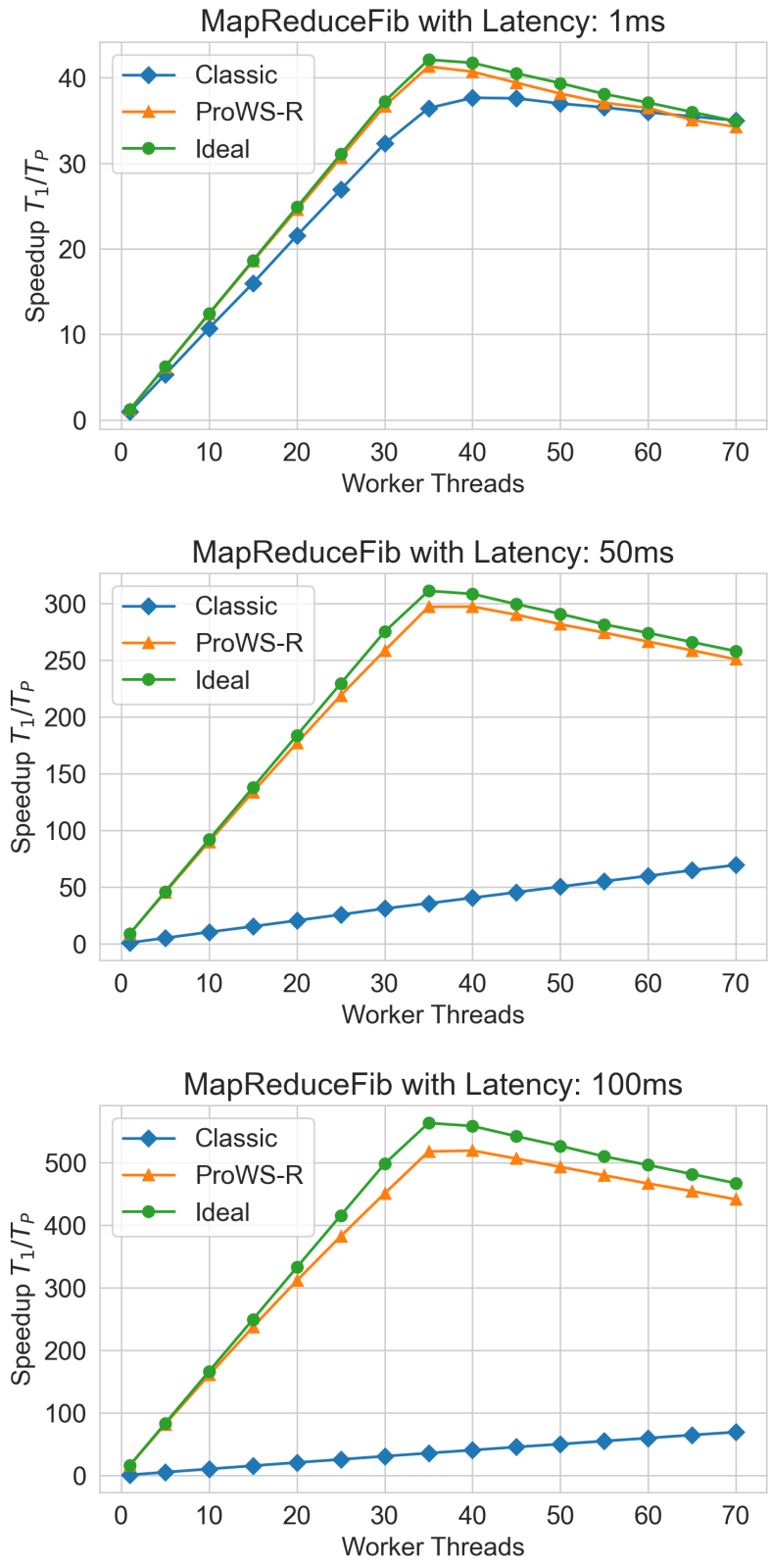

Figure 5.1: MapReduceFib benchmark results with varying latencies

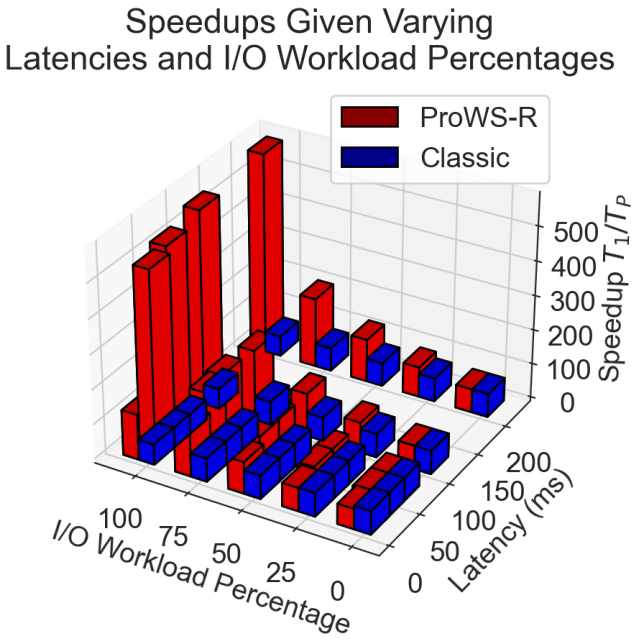

Figure 5.2: Parameter sweep benchmark results

Chapter 6

Conclusion: Patience Is Not a Virtue

6.1 Summary

6.2 Lessons Learned

6.3 Future Work

Bibliography

[1] Advanced HPC Threading: Intel® oneAPI Threading Building Blocks.

[2] Baby Steps.

[3] Does ‘rayon::spawn‘ block until a thread is available? · Issue #522 · rayon-rs/rayon.

[4] futures::future - Rust.

[5] join in futures::future - Rust.

[6] Rayon I/O operations warning.

[7] Rayon-LH Benchmarks.

[8] async-io, April 2022. original-date: 2020-06-28T20:16:29Z.

[9] Crossbeam, April 2022. original-date: 2015-05-13T18:10:54Z.

[10] Rayon, April 2022. original-date: 2014-10-02T15:38:05Z.

[11] Rayon Thread Sleep/Notification Mechanism, April 2022. original-date: 2014-10-02T15:38:05Z.

[12] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA ’98, pages 119–129, New York, NY, USA, June 1998. Association for Computing Machinery.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP ’95, pages 207–216, New York, NY, USA, August 1995. Association for Computing Machinery.

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[15] Robert D. (Robert David) Blumofe. *Executing multithreaded programs efficiently*. Thesis, Massachusetts Institute of Technology, 1995. Accepted: 2005-08-17T23:21:41Z.

[16] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[17] Randal E. Bryant and David R. O'Hallaron. *Computer systems: a programmer's perspective*. Pearson, Boston, third edition edition, 2016.

[18] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, fall joint computer conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, November 1963. Association for Computing Machinery.

[19] Thomas H. Cormen, editor. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 3rd ed edition, 2009. OCLC: ocn311310321.

[20] Mache Creeger. Multicore CPUs for the Masses: Will increased CPU bandwidth translate into usable desktop performance? *Queue*, 3(7):64–ff, September 2005.

[21] Daniel Etiemble. 45-year CPU evolution: one law and two equations. *arXiv:1803.00254 [cs]*, March 2018. arXiv: 1803.00254.

[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, May 1998. Association for Computing Machinery.

[23] Robert H. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17, New York, NY, USA, August 1984. Association for Computing Machinery.

[24] Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[25] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, January 2019.

[26] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, Amsterdam, revised first edition edition, 2012.

[27] Joel. DashMap, April 2022. original-date: 2019-12-06T21:48:39Z.

[28] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, June 2000. Association for Computing Machinery.

[29] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. Conference Name: Computer.

[30] Raspberry Pi Ltd. Raspberry Pi Zero 2 W.

[31] Nicholas D. Matsakis and Felix S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, October 2014.

[32] Stefan K. Muller and Umut A. Acar. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 71–82, New York, NY, USA, July 2016. Association for Computing Machinery.

[33] Eric Niebler. Structured Concurrency, November 2020.

[34] Linus Nyman and Mikael Laakso. Notes on the History of Fork and Join. *IEEE Annals of the History of Computing*, 38(3):84–87, July 2016. Conference Name: IEEE Annals of the History of Computing.

[35] Jeff Parkhurst, John Darringer, and Bill Grundmann. From Single Core to Multi-Core: Preparing for a new exponential. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 67–72, November 2006. ISSN: 1558-2434.

[36] David Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 53, July 2010.

[37] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. RISC-V edition. Elsevier, Cambridge, MA, second edition edition, 2021.

[38] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in HPC systems: Why and when? *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2016.

[39] rodyamirov. How can I mix rayon and tokio properly?, June 2020.

[40] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, Proceedings, pages 147–161. Society for Industrial and Applied Mathematics, December 2019.

[41] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 257–271, New York, NY, USA, February 2019. Association for Computing Machinery.

[42] Nathaniel J. Smith. Notes on structured concurrency, or: Go statement considered harmful — njs blog.

[43] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures - SPAA '09*, page 91, Calgary, AB, Canada, 2009. ACM Press.

[44] Josh Stone. How Rust makes Rayon's data parallelism magical, April 2021.

[45] Neil Weidinger. Rayon-LH, December 2021. original-date: 2021-12-22T16:30:00Z.

[46] Christopher S. Zakian, Timothy A. K. Zakian, Abhishek Kulkarni, Buddhika Chamith, and Ryan R. Newton. Concurrent Cilk: Lazy Promotion from Tasks to Threads in C/C++. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing*, pages 73–90, Cham, 2016. Springer International Publishing.