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 [24, 35]. One such paradigm is ***fork-join*** parallelism [11, 26], 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 [15], the Java Fork/Join framework [20], and TBB [1], commonly employ ***work-stealing*** [6]: 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 [7] as well as proving themselves in practice [5]. 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 [24, 30, 35].

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 [25, 32].

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 [31], introduce a latency-hiding work-stealing scheduling algorithm, ***ProWS***, that utilizes ***futures*** (a parallel language construct [16, 17]) 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 [24, 33].

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 [30] present an extension of Cilk, called Cilk-L, that incorporates such runtime support with encouraging results.

TODO: better transition to talking about Rust

Rust [23] 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 [2, 34] is a widely-used library level implementation (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 [3]. 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 designed 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 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 [18].

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 [9]. 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 [27]. 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 [28].

CPU performance can be described using the following equation [29], 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 [14].

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 [22].

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 [13, 21, 28].

- 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 [28].

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 [21].

[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 [10]. 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 [12, 19, 8]
- 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** 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

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

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 [31], 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*** [33], 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 [24] 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.

$$\begin{tikzcd}
\text{Active} \arrow[r, "\text{Encounter blocked future}"] & \text{Suspended} \arrow[d, "\text{Completion of blocked future}"] \\
\text{Muggable} \arrow[u, "\text{Mugged by thief}"] & \text{Resumable} \arrow[l, "\text{Stolen by thief once before}"]
\end{tikzcd}$$

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.

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 19) 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.

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. More concrete reasoning and explanation of how this occurs in practice is given in section 4.3.

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 [2]: all that is required is a check (line 37) to see if the deque 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.

[1]The `join` function in the Rust futures crate [4] 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.

[2]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 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.

When a worker thread cannot find work to execute in its active deque (line 31), it must become a thief and steal from elsewhere. The steal procedure is outlined in algorithm 3. 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 [31, 30]. 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 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.

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

```
function SCHEDULINGLOOP
    while computation is not done do
        $node \leftarrow$ findNode()
        $left, right \leftarrow$ execute($node$)
        if $left \neq$ null then
            $w$.active.pushBottom($left$)
        end if
        if $right \neq$ null then
            $w$.active.pushBottom($right$)
        end if
        if $node$ encountered blocked future $f$ then
            $deq \leftarrow w$.active
            $w$.active $\leftarrow$ null
            suspendDeque($deq$)
            $f$.installCallBack($deq$)
        end if
    end while
end function
function SUSPENDDEQUE($deq$)
    $deq$.state $\leftarrow$ SUSPENDED
    $w$.stealableSet.remove($deq$)
    if !$deq$.isEmpty() then
        chooseRandomVictim().stealableSet.add($deq$)
    end if
end function
function FINDNODE
    $node \leftarrow$ null
    if $w$.active $\neq$ null then
        $node \leftarrow w$.active.popBottom()
    end if
    if $node =$ null then
        $node \leftarrow$ steal()
    end if
    return $node$
end function
```

Algorithm 2 Callback Procedure (called upon completion of the blocked future f)

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

3.2 Required Runtime Support for Latency-Hiding

The ProWS-R algorithm on its own is not enough to enable latency-hiding [3]. 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 to solely this task. 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 [4]. 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 [5], with the I/O thread not taking

[3] This section is based off the work by Singer et. al on Cilk-L [30], a latency-hiding extension of Cilk that uses the ProWS algorithm, with considerations on how to integrate with the mechanisms involved with Rust futures.

[4] Classic work-stealing runtimes create P worker threads for P physical processor cores, to maximize hardware usage efficiency [5].

[5] 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

Algorithm 3 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
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
```

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:

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 2 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 2) 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 [6] which then executes the callback. This achieves part 2.

---

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 [5].

[6]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 2). 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

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. 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.

the resource it is blocked on with the I/O thread event queue (part 1).

Chapter 4

Implementation: Time is an Illusion

Chapter 5

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

5.1 Experimental Setup

5.2 Impact of Injecting Latency

5.3 Compute vs I/O Bound Workloads

5.4 Limitations

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] futures::future - Rust.

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

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.