Performance Analysis and Debugging of Parallel Programs within Lightweight Scheduling Frameworks

by

Darshan Dinesh Kumar

A thesis submitted in partial fulfillment

Of the requirements for the degree of

Master of Science

Courant Institute of Mathematical Sciences

New York University

May, 2026

Sam Westrick

© Darshan Dinesh Kumar

All rights reserved, 2026

Dedication

To My Family

Acknowledgments

I am extremely grateful and deeply indebted to my advisor, Prof. Sam Westrick, for his invaluable guidance and support throughout this research. Since our initial exploratory meeting in November 2024, he has been a constant pillar of support during my time at NYU. I want to sincerely thank him for giving me the opportunity to collaborate with him. Over the past 17 months, despite his demanding schedule, he consistently dedicated quality time to our weekly meetings—discussing my progress, providing thoughtful feedback, and offering crucial advice on navigating the research process. I am immensely grateful for his mentorship in developing my research skills and guiding me through the complexities of parallel computing; his insights were instrumental in shaping the direction and quality of this thesis. Our conversations, which seamlessly transitioned from deep technical discussions to shared personal anecdotes, have been a true highlight of my master's experience. Thank you, Prof. Westrick, for your unwavering support, your expertise, and the friendship that has grown along the way.

I would also like to extend my sincere gratitude to Seong-Heon Jung, a PhD student advised by Prof. Westrick, and to Prof. Westrick himself, for our collaboration on the Scheduler Augmentation project. What began as a nascent idea while attempting to devise the fluctuation metric quickly evolved into a full-fledged research endeavor. Seong-Heon has been an incredible collaborator, and his crucial role in developing the framework—along with our engaging technical discussions—made this project a highly rewarding experience.

My sincere thanks go to Prof. Aurojit Panda for graciously agreeing to serve as the Second

Reader for this thesis. I am deeply appreciative of his time and his willingness to review my work. Having had the opportunity to interact with him as part of the Distributed Systems course, I am truly thankful for his continued support in this capacity.

I also wish to thank the NYU Courant Department of Computer Science for giving me the opportunity to pursue this research thesis as part of my Master's degree. The resources and support provided by the department have been essential in enabling me to carry out and complete this work.

Finally, I owe an immeasurable debt of gratitude to my family and friends. Their unwavering belief, support, and encouragement have been my foundation, making it possible for me to pursue this Master's degree and embark on this research journey in the first place.

Abstract

The widespread adoption of multi-core architectures has made parallel programming a fundamental paradigm. A popular mechanism to expose primitives for parallelism is through lightweight parallel libraries that require no dedicated compiler support. The majority of these libraries internally implement a work-stealing scheduler to dynamically map parallel tasks onto processors, enabling the programmer to focus on algorithmic concerns and express parallelism at higher levels of abstraction. Because these frameworks are heavily relied upon for high performance, this thesis investigates techniques for the performance analysis and debugging of parallel programs within them. Specifically, we focus on three core areas: granularity analysis using an augmented scheduler, the feasibility of elastic scheduling, and the definition of a structural fluctuation metric.

For performance analysis, optimization, and debugging, it is often useful for a programmer to view the program in a schedule-oblivious manner by focusing on its logical structure in the form of a computation graph, or Directed Acyclic Graph (DAG). However, existing tools for dynamic computation graph analysis are typically tightly integrated with specific compilers and runtime systems, making them difficult to use in conjunction with lightweight libraries. We propose *Scheduler Augmentation* to address this gap by defining a vertex interface to the scheduler that can explicitly track the vertices in the DAG. Users provide their own definitions of vertex-local data and specify how this data is updated at runtime. We demonstrate the power of scheduler augmentation by addressing the granularity control problem using an augmentation-based granularity analysis technique, which identifies subcomputations that are too fine-grained to benefit

from parallelism.

While work-stealing schedulers are highly effective at load-balancing, they can result in significant wasted compute cycles during unsuccessful steal attempts. Elastic scheduling is a mechanism aimed at resolving this by dynamically transitioning idle processors into low-power sleep states to conserve energy. We perform a detailed study to investigate the effectiveness of elasticity by formulating a *Theoretical Feasibility* metric and a visual representation, the *Prefix Sum Plot*, to analyze idle durations and quantify theoretical energy savings. Using these formulated metrics, we evaluate a practical implementation of elasticity in ParlayLib, a lightweight scheduling library for C++, to verify its effectiveness in converting theoretical potential to practical energy savings. Further, we identify a critical race condition and a potential deadlock in its implementation, proposing a “steal-after-sleep-announcement” protocol and a “load-before-decrement” instruction ordering to resolve them.

Our feasibility study reveals that elastic schedulers like ParlayLib exhibit mixed success in converting theoretical potential into practical energy savings. Motivated by these discrepancies, we set out to answer the following research question: *Can we capture and characterize a structural feature of a computation DAG that directly correlates with the scheduler’s ability to extract energy savings?* Specifically, we aim to define an analytical metric—which we term *fluctuation*—capable of classifying DAGs based on their intrinsic suitability for elastic sleep transitions. We explore the proposed fluctuation metric, defined as $F/(W \times S)$, and highlight how, despite providing a nuanced measure of structural variance, it ultimately exhibits significant analytical shortcomings.

Contents

Dedication ii

Acknowledgments iii

Abstract v

List of Figures xi

List of Tables xiii

1 Introduction 1

2 Background 5

2.1 Programming Model: Binary Fork-Join 5

2.2 Analysis: DAGs and Cost Models 6

2.2.1 Series-Parallel DAGs 6

2.2.2 Work-Span Model 6

2.3 Runtime Schedulers and Work-Stealing 7

3 Granularity Analysis using Scheduler Augmentation 9

3.1 Scheduler Augmentation 9

3.1.1 Motivation 9

3.1.2 Vertex Interface 10

3.1.3 Example 11
3.1.4 Implementation Details 12
3.1.4.1 Augmented Work Stealing 12
3.1.4.2 Extending ParlayLib 13
3.2 Granularity Analysis 15
3.2.1 Classifying sub-DAGs by w/f. 16
3.2.2 Case Study: Parallel Range Query 16
3.2.2.1 The PBBS parallel plane sweep benchmark 17
3.2.2.2 Observing a potential granularity control issue. 17
3.2.2.3 Applying granularity analysis 17
3.2.2.4 Investigating and optimizing the PAM GC 19
3.2.2.5 Confirming the optimization 20
3.3 Conclusion 20
4 Investigating the Effectiveness of Elasticity **21**
4.1 Introduction 21
4.2 Feasibility Study 22
4.2.1 Sleep Estimation Experiments 23
4.2.2 Distribution of idle durations 25
4.2.2.1 Quantifying Potential Energy Savings 27
4.2.3 Analyzing idle durations for practical feasibility 28
4.2.3.1 Experimental Setup 28
4.2.3.2 BFS Benchmark 29
4.2.3.3 Primes Benchmark 30
4.2.3.4 Delaunay Benchmark 31
4.2.3.5 Mergesort Benchmarks 32

4.2.3.6 Synthetic Benchmark with Low Parallelism 34
4.2.3.7 Synthetic Benchmark with High Parallelism 37
4.2.3.8 Conclusion 37
4.2.4 Connecting Feasibility Results to actual Elasticity implementation in ParlayLib 38
4.3 Race Condition in the Elasticity Implementation in ParlayLib 41
4.3.1 Reproducing the Race Condition 42
4.3.2 Proposed Solution 44
4.3.3 Execution Order of Loading `wake_up_counter` and Decrementing `num_awake_workers` 45
4.3.3.1 Example Scenario 1 (Ex1): Original Ordering 45
4.3.3.2 Example Scenario 2 (Ex2): Proposed Ordering 46
4.3.4 Correctness of the load-before-decrement order 47
4.3.4.1 ParlayLib's Default Behavior (`atomic_notify_all` fallback) . . 48
4.3.4.2 Strict `notify_one` Implementations (e.g., Win32) 49
4.3.4.3 Conclusion 49
4.3.5 Verification of the Fix 50

5 Towards a definition of Fluctuation **51**
5.1 Motivation 51
5.2 Equivalent Work and Span Measures 53
5.3 Differing Work and Span Measures 54
5.4 Shortcomings and Future Work 55

6 Related Work **57**
6.1 Parallel Programming Models 57
6.2 Parallel Languages 58

6.3 Foundations of Work-Stealing 58
6.4 Profiling Parallel Programs 59
6.5 Scheduler-based Metrics 59
6.6 Lightweight Scheduling Libraries 60
6.7 Cilk Hyperobjects 60
6.8 Granularity Control 60
6.9 Modern Parallel Benchmarking and Frameworks 61
6.10 Energy Efficiency and Waste-Efficient Scheduling 61
6.11 Performance Modeling 62

7 Conclusion **63**

8 References **65**

List of Figures

3.1 Vertex interface. Custom vertex types, provided by the client, must adhere to this interface. 10

3.2 Augmented scheduler interface. Functions `augment` and `getCurrentVtx` are our additions. The `pardo` function is the main fork-join primitive, and remains unchanged. 10

3.3 Scheduler augmentation example, showing an execution of `foo(0,3)`. All omitted code is sequential (no calls to `pardo`). Dashed boxes correspond to nested calls to `foo()`. 12

3.4 The `parlay::pardo` function with augmentation. Our modifications highlighted. 14

3.5 Augmentation to perform granularity analysis by tracking the work and number of forks in every sub-DAG. 15

3.6 Using scheduler augmentation for granularity analysis of every sub-DAG in the PBBS `rangeQuery2d/parallelPlaneSweep` benchmark on 1M points. We measure the work w and number of forks f within each sub-DAG, and plot it at $(w, w/f)$. 18

4.1 Sleep Estimation Experiment 25

4.2 An Example Prefix Sum Plot for the `delaunay` benchmark. 26

4.3 Prefix Sum Plot for the `bfs` benchmark. 30

4.4 Prefix Sum Plot for the `primes` benchmark. 31

4.5 Prefix Sum Plot for the parallel mergesort benchmark. 32

4.6 Prefix Sum Plot for the parallel mergesort with sequential merge benchmark. . . . 33
4.7 Synthetic benchmark with low parallelism. The parallel region is designed such that there is a significant amount of idle time for the threads due to load imbalance. 34
4.8 Prefix Sum Plot for the synthetic low-parallelism benchmark. 35
4.9 Synthetic benchmark with high parallelism. The parallel region is designed such that there is a significant amount of work for the threads to perform, with minimal idle time due to load imbalance. 36
4.10 Prefix Sum Plot for the synthetic high-parallelism benchmark. 36
4.11 Original code for `wait_for_work` and `wake_up_a_worker` functions. 42
4.12 Example code that triggers race condition 43
4.13 Updated code for `wait_for_work` function that resolves the race condition with the changes highlighted. 44
5.1 Two cases of DAG structures under a strict binary fork-join model. 52
5.2 Two computational structures with identical total work ($W = 7$) and span ($S = 5$), but varying degrees of structural fluctuation dictated by the number of forks (F). . 53
5.3 Three diverse general DAG cases demonstrating that distinct computational structures with differing work (W) and span (S) can possess identical F/S ratios ($1/3$ for all), revealing the ratio's insufficiency to model fluctuation. However, the $F/(W \times S)$ values are distinct, $1/12$, $1/36$, and $1/48$ respectively. 54

List of Tables

4.1 Total working, stealing, and sleeping time (in milliseconds) with elasticity OFF, and the raw performance delta (along with proportional ratio) when elasticity is ON for 32 threads. The Theoretical Feasibility values for each benchmark correspond to the computed values for 32 threads as discussed in 4.2.3, while Practical Feasibility represents $1 + (T_{\text{total_sleep}}/T_{\text{total_work}})$ with elasticity ON for 32 threads. 39

4.2 Execution interleaving for the Ex1 Scenario (Original Ordering). 46

4.3 Execution interleaving for the Ex2 Scenario (Proposed Ordering). 47

4.4 Execution interleaving for the Informal Proof Scenario. 47

1 Introduction

The computing landscape has undergone a fundamental transformation over the last two decades. Driven by the plateau of Moore's Law, Dennard scaling, and the physical limitations of single-thread processor frequencies, the hardware industry has universally shifted toward multi-core and many-core architectures. Consequently, parallel programming is no longer a niche domain reserved for supercomputing; it is an essential paradigm required to achieve high performance on everything from mobile devices to massive cloud servers.

To harness this complex hardware efficiently, developers require high-level programming models that expose primitives for parallelism without drowning the programmer in the low-level details of hardware management. Historically, this was achieved either through heavy, custom compilers or dedicated parallel languages. Today, however, a highly popular and practical approach is the use of *lightweight parallel libraries*. These frameworks—such as Intel's Threading Building Blocks (TBB) [1] or ParlayLib [2]—allow programmers to express parallelism entirely within a host language like C++ using standard templates and lambdas. They require no custom compiler support, enabling the developer to focus purely on the algorithmic logic and structural concurrency of their applications.

Beneath the surface, the vast majority of these lightweight libraries rely on a *work-stealing* scheduler to dynamically map parallel tasks onto available processor cores. In a work-stealing runtime, each processor (or worker thread) maintains a local double-ended queue (deque) of tasks. Workers process their own local tasks efficiently; however, when a worker exhausts its

own queue, it becomes a "thief" and actively steals pending tasks from the deques of other busy workers. This abstraction is incredibly powerful: it provides nearly optimal bounds on time, and it automatically handles the complex mechanics of dynamic load balancing.

Given the popularity of these frameworks and the general emphasis on high performance, this thesis investigates techniques for analyzing, profiling, and debugging parallel programs, specifically within lightweight scheduling frameworks. The primary contributions of this thesis and their respective motivations are summarized as follows:

- **Chapter 2: Granularity Analysis using Scheduler Augmentation.** To rigorously analyze and optimize parallel executions, developers benefit significantly from examining the computation abstractly—visualizing the program's logical dependencies as a Directed Acyclic Graph (DAG) rather than tracing OS-level thread assignments. Unfortunately, most existing profiling tools that construct these dynamic graphs are deeply embedded into specialized compilers or heavy runtimes, rendering them incompatible with lightweight, library-based schedulers. To overcome this limitation, we introduce ***scheduler augmentation***. By embedding a customizable *vertex interface* directly into the runtime scheduler, it can naturally expose the unfolding DAG. This framework permits developers to attach custom, localized metrics to the vertices, defining precisely how performance data propagates during fork and join operations.

  We demonstrate the power of scheduler augmentation by addressing the *granularity control problem*. A fundamental tension in parallel execution is determining the appropriate size of tasks. Extracting peak performance requires exposing sufficient parallelism to utilize all hardware threads effectively, naturally driving programmers to aggressively subdivide their workloads. However, managing this parallelism is not free; spawning tasks, handling task queue operations, and synchronizing threads all impose a non-negligible overhead. If a workload is partitioned into excessively small chunks (fine-grained), the system

spends more time executing scheduler logic than performing useful computation, resulting in severe performance degradation. On the other hand, leaving tasks too large (coarse-grained) underutilizes the hardware by creating load imbalances. Balancing this trade-off is incredibly difficult in practice. We present a granularity analysis technique leveraging our augmented scheduler that successfully pinpoints specific subcomputations that are too fine-grained, guiding the programmer on exactly where to coarsen their algorithms.

- **Chapter 3: Investigating the Effectiveness of Elasticity.** In traditional work-stealing scheduling frameworks, the workers are classified as either *working* or *idle*. During non-working intervals, idle workers engage in randomized work-stealing, actively polling the deques of other workers in search of available tasks. Consequently, these idle workers are not energy-neutral; they consume significant CPU cycles and power while busy-waiting. *Elastic scheduling* addresses this inefficiency by introducing dynamic worker management, aiming to transition idle workers into low-power sleep states when no work is immediately available.

  The primary challenge in realizing elastic scheduling lies in executing these state transitions without severely degrading end-to-end application runtime. Because a sleeping worker cannot autonomously detect when new tasks become available, the scheduler must implement a wake-up protocol that guarantees task distribution and preserves scalability. Further, the act of transitioning a thread to and from a sleep state via system calls incurs a tangible latency overhead. Attempting to convert exceedingly brief idle intervals into sleep states is therefore counterproductive, as the transition latency can easily eclipse the theoretical energy savings and inflate the execution time.

  In order to address this, we present a rigorous feasibility study utilizing a novel visual tool, the *Prefix Sum Plot*, alongside a formulated *Feasibility metric*. These tools enable us to accurately quantify the theoretically exploitable idle time and potential energy savings

across various algorithmic benchmarks. We transition from theory to practice by analyzing the elastic scheduler implementation within ParlayLib, a lightweight scheduling library for C++. This allows us to evaluate how effectively the theoretical energy savings can be realized in a state-of-the-art scheduling environment. As a derivative of this work, we identify two critical bugs in the elastic implementation of ParlayLib, an initialization race condition and a possible deadlock due to the ordering of operations. We propose fixes for these by introducing a "steal-after-sleep-announcement" protocol and a robust "load-before-decrement" instruction ordering, respectively.

- **Chapter 4: Towards a Definition of Fluctuation.** Our empirical evaluation of the elasticity implementation in ParlayLib uncovers a critical gap between the anticipated theoretical energy reductions and the actual savings realized during execution. These inconsistencies prompt a fundamental research inquiry: *Can we capture and characterize a structural feature of a computation DAG that directly correlates with the scheduler's ability to extract energy savings?* To investigate this, we formalize a new analytical construct called the *fluctuation* metric. Designed to evaluate a DAG's inherent compatibility with sleep-state transitions, this metric—expressed mathematically as the ratio $F/(W \times S)$—attempts to capture the dynamic variability of available work. We analyze this formulation, demonstrating that while it offers a sophisticated perspective on structural variance, it ultimately possesses inherent limitations that require refining and future work.

2 Background

In this section, we provide the foundational background necessary to reason about the performance and structural characteristics of parallel applications. First, we define the computational paradigm these applications adhere to by introducing the binary fork-join programming model. Next, we discuss the theoretical abstractions and cost models used to analyze them, specifically focusing on Series-Parallel Directed Acyclic Graphs (DAGs) and the Work-Span model. Finally, we detail the underlying runtime mechanisms responsible for mapping these theoretical graphs onto physical hardware, focusing on the mechanics of work-stealing schedulers.

2.1 Programming Model: Binary Fork-Join

We consider parallel computations written in the binary fork-join style. At its core, this programming model can be expressed in terms of a single fundamental primitive: `pardo(f, g)`.

A call to `pardo` performs three distinct operations: (1) it *forks* (spawns) two child tasks to execute the function calls `f()` and `g()` concurrently; (2) it synchronizes by waiting at a *join* point for both child tasks to complete; and (3) it resumes the execution of the calling context. These calls to `pardo` can be freely and recursively nested to express complex parallel structures.

Furthermore, this simple binary primitive serves as the building block for higher-level parallel abstractions. For instance, a parallel for-loop (`parfor`) is typically not implemented as a monolithic spawn of N iterations. Instead, it is implemented recursively using `pardo` by recursively

splitting the loop range in half until a base case (grain size) is reached, at which point the iterations are executed sequentially. This recursive divide-and-conquer approach ensures that the depth of the task creation tree remains logarithmic with respect to the loop bounds.

2.2 Analysis: DAGs and Cost Models

2.2.1 Series-Parallel DAGs

To rigorously analyze these fork-join computations, it is standard practice to use the computation graph abstraction, also called the Directed Acyclic Graph (DAG). In a computation graph, vertices represent discrete units of sequential computation that execute uninterrupted by any forks or joins. The directed edges encode the scheduling dependencies or precedence constraints between these computations; the absence of cycles ensures a valid partial order of execution.

Because binary fork-join parallelism restricts task synchronization to block-structured scopes, it naturally yields a specific class of computation graphs known as *Series-Parallel (SP) DAGs*. These graphs are inductively defined through the serial and parallel composition of subgraphs. Consequently, every valid SP graph possesses a well-defined single *source* vertex and a single *sink* vertex, which could be equal (in the singleton case). A fork corresponds to an outward branching in this graph, while a join represents a synchronization point where the sink vertices of the child tasks converge.

2.2.2 Work-Span Model

While the Random Access Machine (RAM) model is widely used for sequential computing, the *Work-Span* (or Work-Depth) model is the standard cost model for evaluating parallel algorithms via the DAG abstraction. The model is defined by two fundamental metrics:

- **Work (W):** The total number of vertices (or operations) in the DAG, representing the se-

quential runtime of the program on a single processor.

- **Span (S):** The length of the longest dependency path (the critical path) through the DAG, representing the theoretical lower bound on execution time assuming an infinite number of processors.

A key derivative of this model is the theoretical *parallelism* of the algorithm, defined as the ratio of work to span (W/S). This ratio indicates the average amount of work available to be executed concurrently along every step of the critical path.

2.3 Runtime Schedulers and Work-Stealing

While the Work-Span DAG represents the theoretical execution of a program, mapping these tasks to physical processors is the responsibility of a runtime *scheduler*. When a program executes a `pardo`, the scheduler must dynamically manage the spawned tasks and assign them to available processor cores.

The de facto standard for executing such fork-join parallel computations is the *work-stealing* scheduler. In work-stealing, computations run on a set of ***workers***, each with a local double-ended task queue (***deque***) whose ends are referred to as the *top* and *bottom*. At every moment, each worker is either working on a ***current task***, or it is ***stealing*** (and itself has an empty deque). When stealing, the worker repeatedly attempts to steal a task from the top of other workers' deques until it succeeds, and therefore acquires a new current task and begins working. If a task forks, the worker creates the left and right tasks, pushes one onto its deque, assigns the other as its current task, and begins executing it. Whenever a worker completes a task, it checks the status of the sibling. If the sibling is at the bottom of the worker's own deque, the worker pops it and executes it; if the sibling was previously completed on a different processor, the worker proceeds with the task corresponding to the computation after the join-point; otherwise, the worker switches to stealing.

In summary, we have presented the foundational background and relevant definitions necessary to navigate the rest of the Thesis. Given this background, we can now transition to the core contributions of this Thesis regarding the performance analysis and debugging of parallel programs within lightweight scheduling frameworks.

3 Granularity Analysis using Scheduler Augmentation

In this section, we present *Scheduler Augmentation*—a lightweight, customizable, and low-cost profiling technique for fork-join parallel programs. Following the introduction of the framework, we detail a Granularity Analysis technique that leverages this augmentation to isolate excessively fine-grained subcomputations, enabling developers to perform targeted coarsening or pruning.

3.1 Scheduler Augmentation

3.1.1 Motivation

For performance analysis, optimization, and debugging, it is often useful to abstract away the low-level details of how a parallel program is scheduled, and instead focus on the logical structure of the program's execution in the form of a *computation graph* (a.k.a. "the DAG").

However, existing tools for dynamic computation graph analysis are tightly integrated with specific compilers and run-time systems, making them difficult to use in conjunction with lightweight scheduling libraries.

We propose ***scheduler augmentation*** to address this gap. Our approach is to add a *vertex interface* to the scheduler, enabling it to explicitly track vertices in the computation graph. Users then provide their own definitions of vertex-local data and specify how this data should be

```
class V {
  // ... omitted: constructors, etc.
  void start();
  void stop();
  void fork(V* left, V* right);
  void join(V* left, V* right, V* afterJoin);   }
```

Figure 3.1: Vertex interface. Custom vertex types, provided by the client, must adhere to this interface.

```
template <typename LF, typename RF>
void pardo(LF&& left_func, RF&& right_func);

template <typename F>
V augment(F&& func); // indicate region of interest

V* getCurrentVtx(); // can be called inside augmented region
```

Figure 3.2: Augmented scheduler interface. Functions `augment` and `getCurrentVtx` are our additions. The `pardo` function is the main fork-join primitive, and remains unchanged.

updated, locally, at forks and joins.

3.1.2 Vertex Interface

To observe the computation graph during execution, the programmer provides a ***vertex definition*** which satisfies the vertex interface shown in Figure 3.1. Specifically, the programmer defines a vertex type, `V`, and four methods on this type:

`v.start()`: called when the vertex `v` begins execution.

`v.stop()`: called when `v` finishes its execution due to either a fork, join, or program termination.

`v.fork(&vl,&vr)`: called when a vertex `v` forks, creating two child vertices `vl` and `vr`.

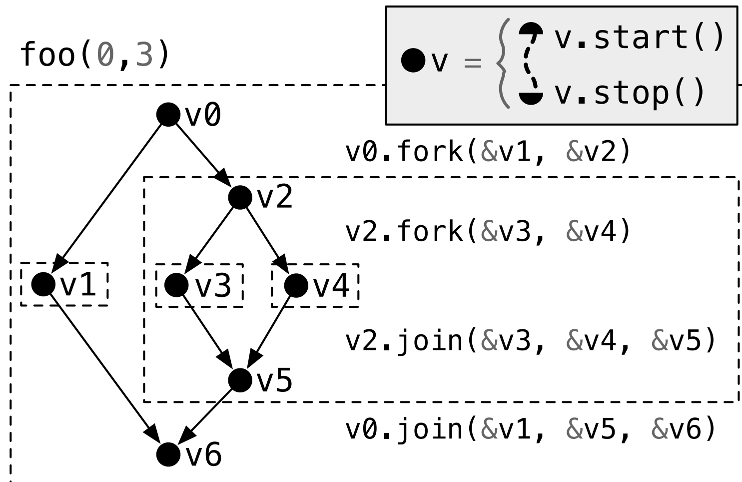
`v.join(&vl,&vr,&vc)`: called when the two vertices `vl` and `vr` join into a new vertex `vc` for the resumed caller. The vertex `v` is the vertex of the corresponding fork.

The SP structure guarantees that every call to `v.fork(&vl,&vr)` has a corresponding call to `v.join(&vl_end,&vr_end,&vc)`, where `vl_end` and `vr_end` are the sink vertices of the two child tasks. These vertices may or may not be equal to `vl` and `vr`, depending on whether or not the children performed any nested calls to `pardo`.

The scheduler implicitly maintains and updates vertices, guaranteeing that all user code executes "at" some current vertex `v` sandwiched between calls to `v.start()` and `v.stop()`. This facilitates lightweight profiling, for example, by starting and stopping a timer for each vertex. The callbacks to `v.fork()` and `v.join()` can be used by the programmer to track the evolution of the computation graph during execution, which has a number of interesting use-cases such as work and span profiling, space profiling, and granularity analysis. Often, all augmentation data can be efficiently updated only at forks and joins, allowing for this data to remain local to vertices and resulting in low cost for a wide variety of augmentations. We also include the `augment` utility function which the programmer may wrap around a region of interest. This initializes a fresh vertex before executing the region and returns the final vertex when the region finishes execution. Within the region, the programmer may also use `getCurrentVtx()` to access vertices at runtime. Figure 3.2 illustrates the augmented scheduler interface, which includes the vertex interface as well as the `augment` utility function and `getCurrentVtx()` function.

3.1.3 Example

Figure 3.3 illustrates scheduler augmentation through a small example function `foo(lo,hi)` which finds a midpoint `mid` between `lo` and `hi`, and recursively executes `foo(lo,mid)` and `foo(mid,hi)` in parallel, with an appropriate base case. The example calls `foo(0,3)`, resulting in two splits, first at `mid=1` and then (on the right-hand side) at `mid=2`. Altogether, executing `foo(0,3)` results in 7 vertices, labeled `v0-v6`, as shown in Figure 3.3. The scheduler automatically creates these vertices and calls the corresponding `start`, `stop`, `fork`, and `join` methods throughout execution.




```
void foo(int lo, int hi) {
 if (hi-lo <= 1) { ...; return; }
 ...; int mid = lo + (hi-lo)/2;
 pardo([&]{ foo(lo, mid); },
       [&]{ foo(mid, hi); });
}
```

```
V v6 = augment([&]{
  // v0 implicitly here
  foo(0,3);
  // now at v6
});
```

Figure 3.3: Scheduler augmentation example, showing an execution of `foo(0,3)`. All omitted code is sequential (no calls to pardo). Dashed boxes correspond to nested calls to `foo()`.

3.1.4 Implementation Details

We present our implementation of scheduler augmentation, based on ParlayLib's work-stealing scheduler. We first describe how to extend the work-stealing algorithm, and then present details of our specific changes to ParlayLib.

3.1.4.1 Augmented Work Stealing

To incorporate scheduler augmentation into work-stealing, it is necessary to keep track of the vertices of the computation graph and how they map onto tasks. To do so, we equip each task with a ***latest vertex***. We say that the ***current vertex*** of a worker is the latest vertex of its current task. The augmented work-stealing algorithm proceeds as follows:

Fork: At current vertex v, if the worker encounters a fork, it first calls `v->stop()`, and then

executes `v->fork(l, r)` with fresh vertices `l` and `r`. It creates a new task for the left and assigns `l` as its latest vertex, and similarly for the right with `r`. It pushes the right task to the bottom of its deque, and begins executing the left task.

Completing a task: At current vertex `v`, if the worker completes a task, it first calls `v->stop()` to mark the end of the current vertex. It then checks the status of the sibling; there are three cases: (1) If the sibling has already completed, the worker proceeds with a Join (see below). (2) If the sibling is at the bottom of the worker's own deque, the worker pops it, calls `start()` on the sibling's latest vertex, sets the sibling as its current task, and begins executing the task. (3) Otherwise, the worker begins stealing.

Join: At current vertex `v` with a completed sibling vertex `u`, the worker identifies the latest vertex `p` of the parent task. The worker executes `p->join(v, u, p')` with a fresh vertex `p'`, sets the latest vertex of the parent task to `p'`, calls `p'->start()`, and resumes the parent task (executing the computation after the join).

Steal: When a worker steals a task, it sets its current vertex to the latest vertex `v` of the acquired task, executes `v->start()`, and then begins executing the task.

3.1.4.2 Extending ParlayLib

We now walk through how we concretely implement scheduler augmentation in ParlayLib, with details shown in Figure 3.4. Our implementation follows the high-level design described in Section **??**, and integrates these into the `pardo` function of ParlayLib.

We track each worker's current vertex by a thread local vertex pointer (line 1), since ParlayLib allots exactly one `pthread` per worker. Additionally, we add a vertex pointer field to ParlayLib's task object (`WorkStealingJob`) for storing its latest vertex.

The vertices themselves reside in the stack of `pardo`. At a fork, we create the initial vertices for the left and right subtasks on the stack of the parent task (the caller of `pardo`). The subtasks then reuse this address when updating its latest vertex. We see this in the synchronization process.

```
1 static thread_local Vertex* currV;
2 template<typename L, typename R>
3 void pardo(L&& lf, R&& rf) {
4   currV->stop();
5   Vertex* parentV = currV;
6   Vertex leftV, rightV;
7   WorkStealingJob rightTask = makeJob(rf, &rightV);
8   parentV->fork(&leftV, &rightV);
9   deques[myWorkerId()].pushBot(&rightTask);
10  currV = &leftV; leftV.start();
11  lf(); // subtasks may overwrite leftV/rightV
12  leftV.stop();
13  if (deques[myWorkerId()].tryPopBot()) { // unstolen
14    currV = &rightV; rightV.start();
15    rf();
16    rightV.stop();
17  } else { // stolen
18    waitUntil(rightTask.finished());
19  }
20  Vertex afterV;
21  parentV->join(&leftV, &rightV, &afterV);
22  *parentV = std::move(afterV);
23  currV = parentV; currV->start();
24 }
```

Figure 3.4: The `parlay::pardo` function with augmentation. Our modifications highlighted.

First, a new `afterV` vertex is created (line 20). After calling `join` on `parentV`, we `std::move` the `afterV` into the address of `parentV`, effectively overwriting it (line 22).

The safety of this implementation hinges on two quirks of ParlayLib. Contrary to other frameworks, ParlayLib does not expose fork and join primitives, instead handling them both inside the `pardo` function. This ensures the stack frame of `pardo` strictly outlives the child tasks and makes their vertices safe to allocate on the stack. The root vertex is the only exception, as it is created at program initialization (and hence before `pardo`). So, we manually create and free the root vertex at scheduler construction and destruction.

```
class GrainAnalysisVertex {
  double w = 0.0;  int64_t f = 0;  time_t t;  int p;
  void setPhase(int phase) { p = phase; }
  void start() { t = time::now(); }
  void stop() { w += (time::now() - t); }
  void fork(V* l, V* r) { f++; l->p = r->p = p; }
  void join(V* l, V* r, V* c) {
    c->p = p; c->w = w + l->w + r->w; c->f = f + l->f + r->f;
    if (c->w > THRESHOLD) writeLog(p, c->w, c->f);   }}
```

Figure 3.5: Augmentation to perform granularity analysis by tracking the work and number of forks in every sub-DAG.

3.2 Granularity Analysis

A fundamental challenge in writing efficient fork-join parallel programs is the *granularity control problem.* To achieve maximum theoretical performance, a program must expose enough parallelism to keep all available processors busy, which encourages aggressively subdividing work into smaller parallel tasks. However, task creation, deque management, and synchronization incur a non-negligible overhead. If work is divided too far (becoming excessively *fine-grained*), the runtime overhead of the scheduler begins to dominate the actual useful computation, leading to severe slowdowns. Conversely, if tasks are left too large (*coarse-grained*), the program risks starving available cores and suffering from load imbalance. The granularity control problem is the engineering challenge of finding the optimal threshold—the grain size—where enough parallelism is exposed without drowning the system in scheduling burden.

Solving this problem is notoriously difficult in practice. Problematic subcomputations are often dynamically generated, deeply buried within within a complex codebase, for example, hidden within a library that the programmer did not author.

To address this, we present a granularity analysis techique that leverages scheduler augmentation to identify subcomputations that are too fine-grained and thus could benefit from coarsening or pruning. Our technique provides a way to quickly discover these subcomputations, without

any major changes to the code with minimal effort by the programmer.

3.2.1 Classifying sub-DAGs by w/f.

The key idea behind this granularity analysis technique is to collect information about every sub-DAG of the computation graph, and classify these sub-DAGs by how coarse-grained (or fine-grained) they are. In particular, we propose classifying sub-DAGs by the ratio w/f, where w is the amount of work performed within the sub-DAG, and f is the number of forks within the sub-DAG. Intuitively, this ratio can be then be compared against the (known) cost of a single call to `pardo`, which we call τ. (On our machine, we measure $\tau \approx 50$ns) If $w/f \gg \tau$, then the sub-DAG is sufficiently granularity-controlled; if $w/f \approx \tau$ (say, within the same order-of-magnitude), then the sub-DAG is too fine-grained.

Calculating w and f for each sub-DAG using scheduler augmentation is relatively straightforward, as shown in Figure 3.5. We associate with each vertex four fields: a work counter w, a fork counter f, a timestamp t, and a "phase" identifier p (which is useful for our case study and is discussed in Section 3.2.2, below). Each vertex tracks its own work using a timer in the `start` and `stop` methods, and updates the w and f counters appropriately at forks and joins. At each join point, we calculate the ratio w/f and append this information to a log, if the ratio exceeds some threshold (to amortize the cost of logging itself). After the computation finishes, we can then analyze the log to identify sub-DAGs that have small w/f ratios, and are thus too fine-grained.

3.2.2 Case Study: Parallel Range Query

To demonstrate the effectiveness of this technique, we apply it to the `rangeQuery2d/parallelPlaneSweep` benchmark from the Problem-Based Benchmark Suite [3]. Our investigation of this benchmark was motivated by an observation that the performance of the benchmark appeared to be highly sensitive to small changes to the scheduler, which hints at a granularity control issue. As part of

this process, we developed a visualization technique to help analyze the results of the granularity analysis, which we present here.

3.2.2.1 The PBBS parallel plane sweep benchmark

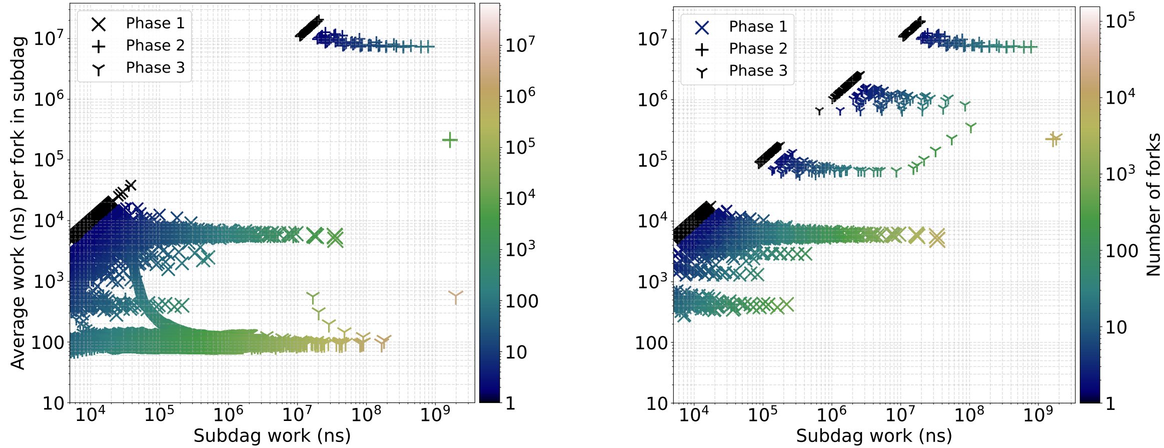
The high-level structure of the algorithm consists of three phases. (1) First, it builds one range-tree at each x-coordinate by sweeping across the points. The range trees are provided by the PAM library [4]. (2) Second, it performs the queries in parallel (specifically by binary searching to identify the relevant left and right range trees, splitting these trees at the top and bottom query range, and taking the difference in sizes). (3) Finally, it deallocates all of the range trees, which triggers a reference-counting garbage collector within PAM. The benchmark waits for the GC to finish before returning.

3.2.2.2 Observing a potential granularity control issue.

In our experiments, we noticed that this benchmark was highly sensitive to the cost of the `pardo` function. In particular, artificially inflating the cost of `pardo` by a small amount caused a significant slowdown ($\approx 20\%$) in the overall runtime. This led us to suspect that the benchmark contains a significant subcomputation that is too fine-grained. However, it is not immediately obvious where to look to find the offending subcomputation. Most of the complexity of the benchmark is offloaded to the PAM library, which handles parallel traversals and manipulations of the range tree data structures.

3.2.2.3 Applying granularity analysis

To investigate, we applied our granularity analysis technique, which in this case consists of (1) augmenting the scheduler with the vertex definitions shown in Figure 3.5, (2) adding three lines of code to the benchmark source, calling `getCurrentVtx->setPhase()` immediately before the start of each phase of the algorithm, to help track which phase each vertex appears within,



(a) Original benchmark: a large number of sub-DAGs in the 3rd phase perform only 100ns of work per fork on average.

(b) After incorporating a simple granularity control heuristic: the 3rd phase of the algorithm is now coarsened.

Figure 3.6: Using scheduler augmentation for granularity analysis of every sub-DAG in the PBBS `rangeQuery2d/parallelPlaneSweep` benchmark on 1M points. We measure the work w and number of forks f within each sub-DAG, and plot it at $(w, w/f)$.

and finally (3) compiling and running the benchmark. This produces a trace of entries (p, w, f), indicating that in phase p of the algorithm there was a sub-DAG with work w and number of forks f.

To analyze the resulting trace, we can plot the entries (p, w, f) on a scatter plot, as shown in Figure 3.6(a).

Here, we plot each sub-DAG at position $(w, w/f)$, where w and f are (respectively) work and number of forks performed within the sub-DAG. Such a plot has the following characteristics.

- **The granularity of each sub-DAG** is represented by its y-coordinate: higher is coarser, and lower is finer.

- **The impact of each sub-DAG** on the overall runtime of the computation is represented by its x-coordinate: larger is more impactful, smaller is less impactful.

Immediately, we observe a cluster (near the bottom of the plot) of sub-DAGs that are very fine-grained, performing only approximately 100ns of work per fork on average. In comparison to the

cost of a fork ($\tau \approx$ 50ns), this ratio of $w/f \approx$ 100ns suggests that as much as 1/3 of the total cost of this portion of the benchmark can be attributed to the overhead of task creation, management, and synchronization. In other words, the benchmark contains a significant subcomputation that is too fine-grained. By tracking the phase of each vertex, we can easily see that all of the fine-grained sub-DAGs appear to be located somewhere within the 3rd phase of the algorithm. This analysis immediately directs us towards the PAM GC implementation.

3.2.2.4 Investigating and optimizing the PAM GC

Looking at the PAM GC call in the benchmark source leads to a recursive function in the PAM source code, called `decrement_recursive`. This function deallocates a PAM tree node by decrementing an associated reference count; if the reference count hits zero, it frees the node and then recursively deallocates the node's two children in parallel (using `pardo`). To control for granularity, PAM uses a heuristic: the recursive calls are executed in parallel only if the size (i.e., number of nodes within) the left[1] child is above some constant threshold. This constant threshold guarantees that the cost of the parallelism is amortized only if the reference counts of the descendant nodes are all 1. If any descendant node has a reference count larger than 1, the recursive call at that node will only perform a single decrement and then immediately exit, which is not enough work to amortize the call to `pardo`. To address this, we adjusted the heuristic to check the reference counts of the children and grandchildren of the current node, and only execute the recursive calls in parallel if all of these reference counts are 1. This guarantees that each call to `pardo` is amortized against at least a modest amount of work, to process the immediate neighborhood below the current node.

[1]PAM trees are balanced, so the choice of left versus right is arbitrary.

3.2.2.5 Confirming the optimization

After applying the optimization described above, we then reran our granularity analysis. The resulting plot is shown in Figure 3.6(b). We can see that the cluster of fine-grained sub-DAGs has been completely eliminated, and the new w/f ratio for the 3rd phase of the algorithm has increased substantially, up to $100\mu s$ or more, which is plenty of work to amortize the cost of the parallelism.

3.3 Conclusion

We have presented scheduler augmentation, a technique that enables the programmer to directly observe the computation graph of a parallel execution by providing vertex-local data definitions to the scheduler. The technique is applicable even in library-only approaches, with no special compiler support. Further, we present a granularity analysis technique that leverages scheduler augmentation to identify subcomputations that are too fine-grained and can benefit from pruning or coarsening. The presented case study demonstrates the practicality of our granularity analysis technique for the Parallel Range Query benchmark. The technique's effectivess is confirmed by the coarsening of fine-grained sub-DAGs in the 3rd phase of the algorithm after applying the granularity control heuristic.

4 Investigating the Effectiveness of Elasticity

4.1 Introduction

In traditional work-stealing scheduling frameworks, the workers are classified as either *working* or *idle*. During non-working intervals, idle workers engage in randomized work-stealing, actively polling the deques of other workers in search of available tasks. Consequently, these idle workers are not energy-neutral; they consume significant CPU cycles and power while busy-waiting. *Elastic scheduling* addresses this inefficiency by introducing dynamic worker management, aiming to transition idle workers into low-power sleep states when no work is immediately available.

The primary challenge in realizing elastic scheduling lies in executing these state transitions without severely degrading end-to-end application runtime. Because a sleeping worker cannot autonomously detect when new tasks become available, the scheduler must implement a wake-up protocol that guarantees task distribution and preserves scalability. Further, the act of transitioning a thread to and from a sleep state via system calls incurs a tangible latency overhead. Attempting to convert exceedingly brief idle intervals into sleep states is therefore counterproductive, as the transition latency can easily eclipse the theoretical energy savings and inflate the execution time.

To navigate these challenges, this chapter investigates both the theoretical potential and practical reality of elastic scheduling. The primary objectives and contributions of this chapter are organized as follows:

- **Theoretical Feasibility Analysis:** We present a feasibility study utilizing a novel visual tool, the *Prefix Sum Plot*, alongside a formulated *Feasibility metric*. These tools enable us to accurately quantify the theoretically exploitable idle time and potential energy savings across various algorithmic benchmarks.
- **Practical Evaluation in ParlayLib:** We transition from theory to practice by analyzing the elastic scheduler implementation within ParlayLib, a lightweight scheduling library for C++. This allows us to evaluate how effectively the theoretical energy savings can be realized in a state-of-the-art scheduling environment.
- **Addressing Implementation Challenges:** We identify two critical bugs in the elastic implementation of ParlayLib, an initialization race condition and a possible deadlock due to ordering of operations. We propose fixes for these by introducing a "steal after sleep announcement" protocol and a robust "load-before-decrement" instruction ordering respectively.

4.2 Feasibility Study

The objective of this feasibility study is to characterize the non-working durations—specifically, periods of unsuccessful stealing prior to a successful task acquisition—and determine if these intervals are sufficiently long to justify transitioning the worker to a sleep state. The goal here is to maximize energy savings without incurring a detrimental impact on overall execution time.

4.2.1 Sleep Estimation Experiments

Evaluating the viability of sleep transitions requires an accurate baseline measurement of the overhead associated with putting a thread to sleep and subsequently waking it up. If typical non-working durations are shorter than this state-transition overhead, elasticity is impractical. Conversely, if a substantial proportion of idle periods exceed this threshold, sleep-based energy savings become viable. To quantify this overhead, we designed an experiment (illustrated in Figure 4.1) consisting of the following sequence:

1. An auxiliary thread (`th1`) is spawned and pinned to a specific core (say, core 6). It executes a sleep routine (`my_sleep`), yielding execution by waiting on an atomic synchronization variable (`flag1`).
2. The main thread is pinned to a different core (say, core 0) and simulates a continuous workload via a long sequential compute loop.
3. Upon completing the simulated work, the main thread triggers a wake-up routine (`my_wakeup`), which asserts `flag1` and notifies `th1`.
4. The main thread then busy-waits until `th1` asserts a secondary variable (`flag2`), signaling that `th1` has fully resumed execution and exited its sleep state.
5. The total latency between the main thread initiating `my_wakeup` and the assertion of `flag2` is recorded as the wake-up duration.

```
#define NUM_ITER 1000000

std::atomic<bool> flag1{false};
std::atomic<bool> flag2{false};

void* my_sleep(void *arg) {
    flag1.wait(false);
    flag2.store(true);
}
```

```
void my_wakeup() {
    flag1.store(true);
    flag1.notify_one();
}

double seq_loop(double num) {
    double acc = 0, prod = 0;
    for(int i = 0; i<NUM_ITER; ++i) {
        prod = num*i;
        acc = acc + prod;
    }
    return acc;
}

int main() {

    // pinning the main thread to core 0
    cpu_set_t main_cpuset; CPU_ZERO(&main_cpuset); CPU_SET(0, &main_cpuset);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &main_cpuset) != 0) {
        std::cerr << "Warning: Failed to pin main thread to Core 0\n";
    }

    // creating and pinning th1 to core 6
    pthread_t th1; pthread_attr_t attr; cpu_set_t cpuset;
    pthread_attr_init(&attr);
    CPU_ZERO(&cpuset); CPU_SET(6, &cpuset);
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpuset);

    int result = pthread_create(&th1, &attr, my_sleep, nullptr);
    if (result != 0) {
        std::cerr << "Failed to create thread\n";
        return 1;
    }

    // simulating work
    seq_loop(100000);

    auto start_time = std::chrono::steady_clock::now();
    my_wakeup();
    while(flag2.load() == false) {
        // busy waiting
    }
    auto end_time = std::chrono::steady_clock::now();

    pthread_join(th1, nullptr);
    pthread_attr_destroy(&attr);

    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time -
    ↪  start_time);
```

```
    auto dur_c = duration.count();
    std::cout << dur_c*2 << "\n";

    return 0;
}
```

Figure 4.1: Sleep Estimation Experiment

This experiment isolates and measures the wake-up latency. Assuming the latency to enter the sleep state is similar to the wake-up latency, we approximate the total end-to-end sleep transition overhead as twice the measured wake-up time. To ensure statistical significance, the experiment was repeated 30 times, yielding an average estimated sleep-transition overhead of approximately 45 microseconds on the `crunchy5` CIMS server, which is the same server used for the subsequent feasibility analysis experiments, and its specifications are provided in Section 4.2.3.1.

This 45-microsecond baseline is subsequently utilized as the threshold in our prefix sum distributions to analyze whether observed non-working durations are long enough to justify sleep transitions.

4.2.2 Distribution of idle durations

An *idle duration* is defined as the elapsed time between the end of a worker's i-th working phase and the start of its $(i + 1)$-th working phase. This interval captures the time a worker spends in both the stealing and sleeping states. To support the feasibility analysis of elastic scheduling, we introduce a novel visual representation: the *Prefix Sum Plot*. This plot visualizes the cumulative distribution of *idle durations*—across executions with varying thread counts (1, 2, 4, 8, 16, and 32). In the Prefix Sum Plot, the x-axis represents individual idle durations (in nanoseconds) sorted in ascending order. The y-axis represents the cumulative (prefix) sum of these durations up to the corresponding x-value. Given the high volume and broad range of recorded idle durations, this cumulative approach prevents visual clutter and clearly depicts the distribution of non-working time across the system.

To appropriately contextualize the magnitude of this non-working time, the cumulative sum on the y-axis is normalized by W_1, the total working time of the benchmark executing sequentially (with 1 thread). Consequently, the maximum y-value (the endpoint of the curve) represents the ratio of total idle time to the sequential execution time, effectively illustrating the relative overhead of idleness for that specific thread count. Note that the geometric area under the Prefix Sum curve holds no physical or mathematical significance; the plot serves solely to visualize the distribution and cumulative impact of idle intervals. Additionally, a vertical dotted red line is overlaid on each subplot to denote the *sleep threshold* - the state-transition (sleep) overhead estimated via the prior sleep estimation experiments.

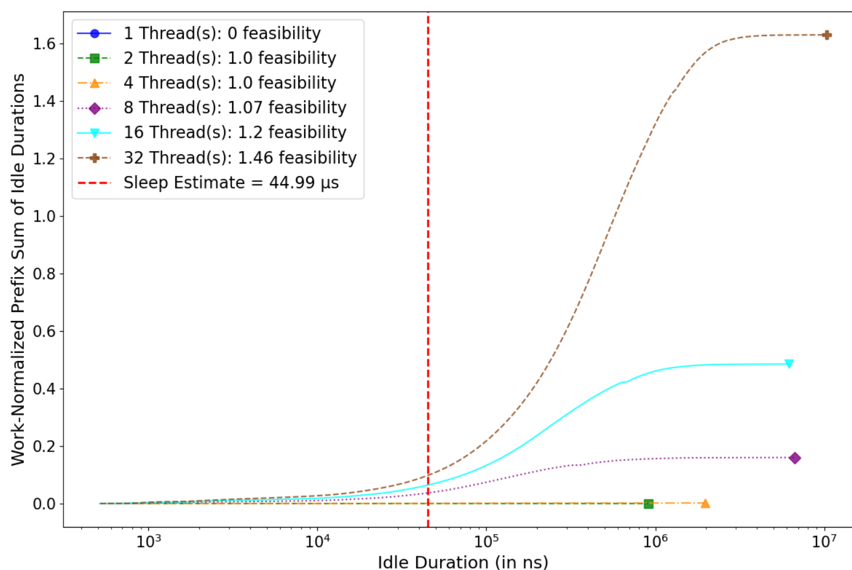


Figure 4.2: An Example Prefix Sum Plot for the `delaunay` benchmark.

Figure 4.2 illustrates an example Prefix Sum Plot for the `delaunay` benchmark, demonstrating how to visually interpret a workload's potential for energy savings. To read the plot, one must observe the behavior of the curves relative to *sleep threshold*: any upward slope to the left of the red line represents idle intervals too brief to justify a state transition, whereas upward slopes to the right represent "Feasible Idle Duration" (FI). For example, executions with 1, 2, and 4 threads show virtually flat lines, indicating minimal idleness and yielding a baseline Feasibility of 1.0.

Conversely, the 32-thread execution exhibits a curve that rises steeply strictly to the right of the threshold, eventually reaching a normalized maximum above 1.6. This visualizes two key facts: total idle time is more than 1.6 times the baseline sequential work time, and the vast majority of this accumulation stems from intervals long enough to comfortably hide the transition overhead. Consequently, this translates to the high Feasibility score (which we define in 4.2.2.1) of 1.46 shown in the legend, indicating a substantial theoretical potential for energy savings.

4.2.2.1 Quantifying Potential Energy Savings

To analytically capture the potential energy savings for a particular benchmark, we define a *Feasibility* metric as follows:

$$\text{Feasibility} = \frac{W_p + \text{TI}}{W_p + (\text{TI} - \text{FI})} \tag{4.1}$$

where:

$$\text{FI (Feasible Idle Duration)} = \sum_{i:\, d_i > S} (d_i - S) \tag{4.2}$$

$$\text{TI (Total Idle Duration)} = \sum_{i=1}^{n} d_i \tag{4.3}$$

- W_p denotes the total working time of the benchmark utilizing p threads.
- d_i denotes the duration of the i-th idle interval.
- S denotes the estimated sleep duration (the overhead to put a worker to sleep and wake it up).
- n denotes the total number of idle intervals.
- The summation $\sum_{i:\, d_i > S}$ ranges exclusively over intervals where the idle duration strictly exceeds the sleep overhead.

The feasibility metric quantifies the theoretical maximum energy savings achievable by putting workers to sleep during non-working durations. It assumes an idealized scenario with perfect oracle knowledge, where a worker is transitioned to a sleep state for every idle interval that is longer than the transition overhead S.

The numerator, $W_p + \text{TI}$, represents the total end-to-end processor time consumed during a standard execution without elasticity. The denominator, $W_p + (\text{TI} - \text{FI})$, represents the total processor time in an idealized elastic execution, where feasible idle time is successfully converted into zero-cost sleep (discounting the required transition overhead S). Thus, the Feasibility metric functions as a ratio comparing the standard execution cost against the idealized elastic execution cost. A higher value indicates a greater potential for energy savings.

In an optimal case where every idle duration strictly exceeds the sleep threshold ($d_i > S$ for all i), the Feasible Idle Duration approaches the Total Idle Duration (FI $\approx$ TI). Under these conditions, the Feasibility metric reaches its theoretical maximum of $1 + \frac{\text{TI}}{W_p}$. As previously established, the normalized endpoint of the Prefix Sum Plot equates to $\frac{\text{TI}}{W_1}$. Assuming parallel workloads generally incur overhead such that $W_p \geq W_1$, the quantity $1 + \frac{\text{TI}}{W_1}$ provides a visual and mathematical upper bound for the Feasibility metric directly deducible from the generated plots.

4.2.3 Analyzing idle durations for practical feasibility

Leveraging the prefix sum plots and the proposed Feasibility metric, we now evaluate the idle duration distributions across various benchmarks to assess the practical viability of sleep-state transitions during non-working periods.

4.2.3.1 Experimental Setup

To empirically analyze these idle times, we conducted our experiments on an `x86_64` NYU CIMS server (`crunchy5`). The server features four AMD Opteron 6272 processors, each containing 16 cores, yielding a total of 64 physical cores. The system does not support simultaneous multi-

threading (SMT), providing strictly one hardware thread per core. The architecture utilizes a Non-Uniform Memory Access (NUMA) design with 8 NUMA nodes and provides 251 GB of main memory. The cache hierarchy comprises private L1 caches per core, 64 MB of L2 cache, and 48 MB of shared L3 cache. The processors support 64-bit execution and vector extensions up to AVX. We executed all experiments on Red Hat Enterprise Linux 9.7 (Plow) and compiled the code using `GCC 11.5.0`.

We executed each benchmark across varying thread counts (1, 2, 4, 8, 16, and 32). During execution, we recorded high-resolution timestamps for the start and end of every working phase. We subsequently used these timestamps to compute the idle durations, generate the corresponding Prefix Sum Plots, and calculate the Feasibility metric to quantify potential energy savings.

It is important to note that we recorded all metrics over the end-to-end execution of the benchmarks. Consequently, they capture the cumulative effect of all non-working durations from the initial scheduler invocation through program termination. This encompasses the startup phase, input data generation, core algorithmic execution, and the shutdown phase. For all evaluations, we utilized the default memory configuration of `ParlayLib` and explicitly disabled its native elasticity features to strictly isolate our measurements.

For our experimental workload, we combined standard benchmarks from the `ParlayLib` suite with a few synthetically generated benchmarks designed to stress specific scheduling behaviors.

4.2.3.2 BFS Benchmark

We evaluated the parallel breadth-first search (`bfs`) benchmark from the `ParlayLib` suite, utilizing an input graph generated via the R-MAT graph generator with 1 million vertices and 20 million edges. Figure 4.3 illustrates the Prefix Sum Plot for this execution. For 32 threads, the normalized endpoint of the plot reaches approximately 0.05, indicating that the total idle time is merely 5% of the sequential (1-thread) working time. The Feasibility metric peaks at 1.04 for 32 threads, suggesting a maximum potential energy savings of roughly 4% if workers are transitioned to

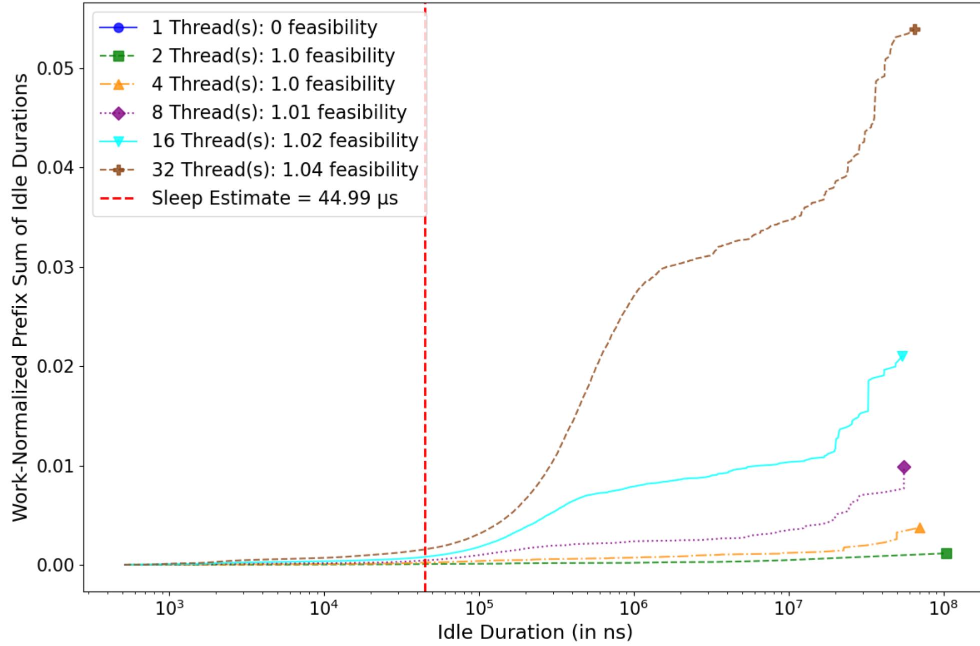


Figure 4.3: Prefix Sum Plot for the `bfs` benchmark.

a sleep state during non-working durations. For lower thread counts, the potential for energy savings is negligible, with Feasibility values remaining near 1.0. Furthermore, even the theoretical maximum Feasibility—calculated as 1.05 (1 + 0.05) under the idealized assumption that total idle time equals feasible idle time (TI = FI)—remains low. Ultimately, these results demonstrate that the `bfs` benchmark exhibits high parallelism; non-working durations are exceedingly short relative to active working time, severely limiting the viability of sleep-based energy savings.

4.2.3.3 Primes Benchmark

We evaluated the parallel prime number generation benchmark from the `ParlayLib` suite, which computes all prime numbers up to a specified integer n. For this experiment, we set n to 10 million. As depicted in Figure 4.4, the normalized endpoint of the Prefix Sum Plot at 32 threads is approximately 0.06, meaning total idle time constitutes roughly 6% of the sequential working time. The Feasibility metric reaches a maximum of 1.04 at 32 threads, translating to a potential energy savings of 4%. Similar to the `bfs` workload, energy savings at lower thread counts are

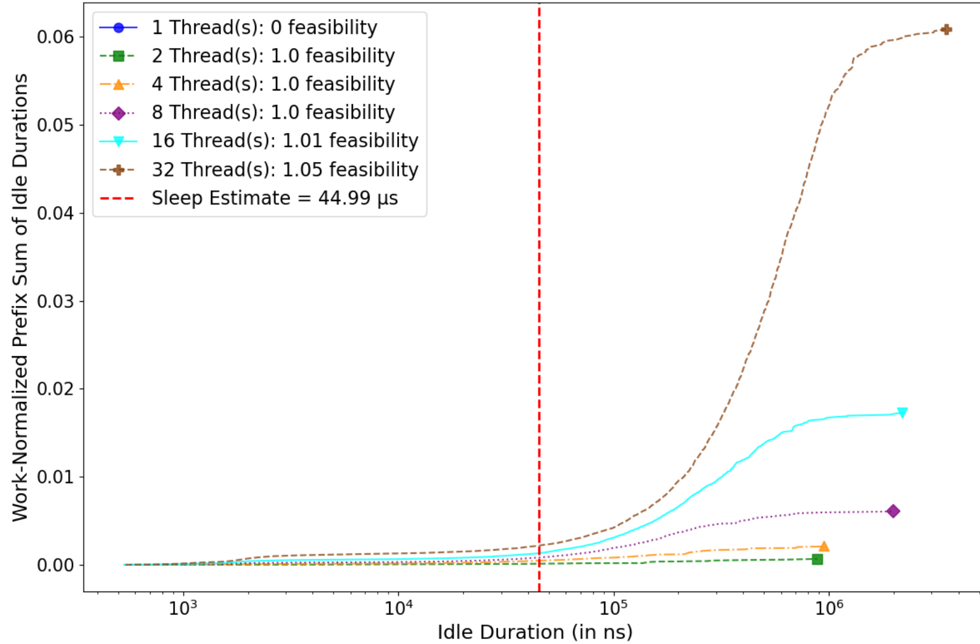


Figure 4.4: Prefix Sum Plot for the `primes` benchmark.

practically non-existent, with the Feasibility metric hovering around 1.0. The theoretical upper bound for Feasibility is similarly constrained at 1.06 (1 + 0.06) assuming TI = FI. These findings reinforce that the `primes` benchmark is highly parallel; its minimal non-working durations render sleep-state transitions largely ineffective for reducing energy consumption.

4.2.3.4 Delaunay Benchmark

We evaluated the parallel Delaunay triangulation benchmark from the `ParlayLib` suite. For this experiment, we generated 1 million random points uniformly distributed within a unit square. Figure 4.2 presents the Prefix Sum Plot for the `delaunay` benchmark. In contrast to the previous benchmarks, the normalized endpoint for 32 threads reaches approximately 1.7, indicating that the total idle time is 170% of the sequential working time. The Feasibility metric captures this behavior, peaking at 1.46 for 32 threads. This suggests a substantial potential for energy savings—up to 46%—if workers are transitioned to sleep during non-working durations. While the potential for energy savings remains negligible for lower thread counts (1, 2, 4, and 8, where Feasibility

values stay near 1.0), it scales significantly as thread count increases, reaching approximately 21% savings at 16 threads. Overall, these results demonstrate that the `delaunay` benchmark is not as parallel as `bfs` or `primes`. The non-working durations are comparatively long, particularly at higher thread counts, presenting a prime opportunity for elastic scheduling to exploit these idle periods for significant energy conservation.

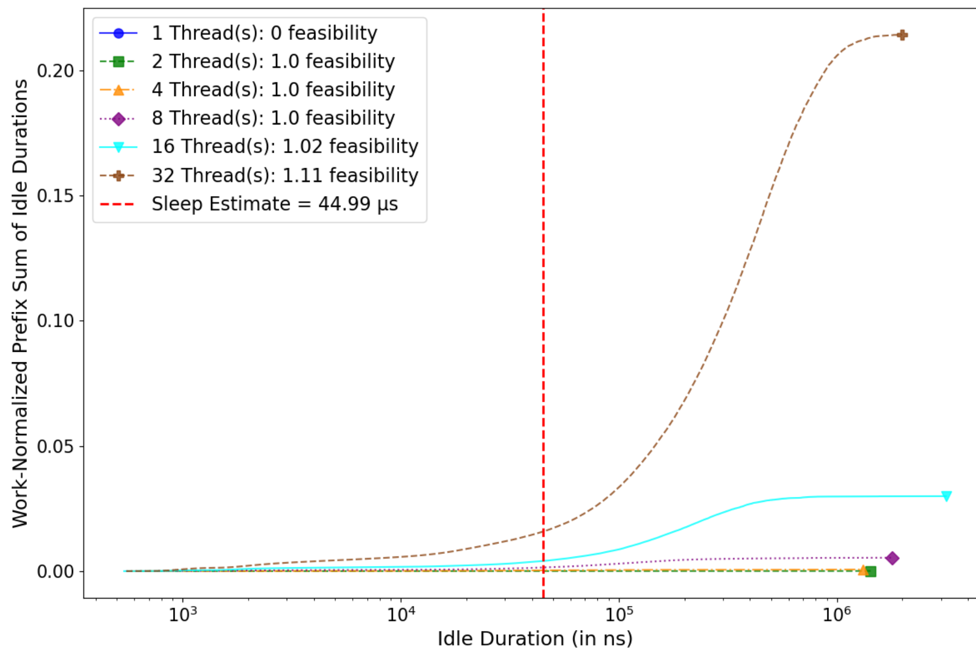


Figure 4.5: Prefix Sum Plot for the parallel mergesort benchmark.

4.2.3.5 Mergesort Benchmarks

In these experiments, we evaluated two variants of the mergesort algorithm, both sorting an array of 10 million integers generated uniformly at random within the range $[0, 10^7)$. The first variant, which we designate as `par_mergesort`, is the unmodified, fully parallel mergesort provided by the `ParlayLib` benchmark suite. The second variant, `seq_merge`, is our custom modification of the `ParlayLib` implementation, wherein we intentionally forced the critical merge step to execute sequentially to observe the effects of a severe structural bottleneck.

Figures 4.5 and 4.6 present the Prefix Sum Plots for the `par_mergesort` and `seq_merge` bench-

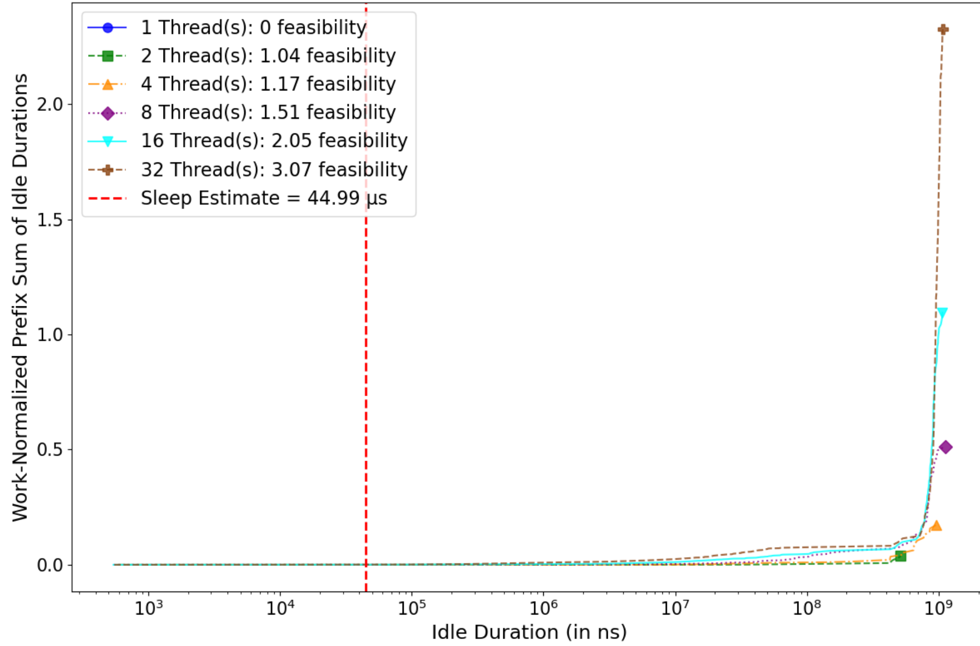


Figure 4.6: Prefix Sum Plot for the parallel mergesort with sequential merge benchmark.

marks, respectively. For `par_mergesort` at 32 threads, the normalized endpoint of the plot is approximately 0.21, indicating that the total idle time is 21% of the sequential working time. Its Feasibility metric reaches a maximum of 1.11 at 32 threads, translating to a potential energy savings of roughly 11% if workers are transitioned to sleep. At lower thread counts (1, 2, 4, and 8), the potential for energy savings remains negligible (with Feasibility values near 1.0), seeing only a minor increase to roughly 2% savings at 16 threads.

Conversely, the sequential bottleneck in our `seq_merge` variant drastically alters the idle time distribution. At 32 threads, the normalized endpoint surges to approximately 2.5, demonstrating that total idle time constitutes 250% of the sequential working time. Consequently, the Feasibility metric for `seq_merge` peaks at a remarkable 3.29 for 32 threads, indicating a massive potential for up to 229% energy savings relative to the standard execution cost. Furthermore, `seq_merge` exhibits a high potential for energy savings across all evaluated thread counts, scaling consistently as the number of threads increases.

Overall, these results validate that the original `par_mergesort` efficiently parallelizes the work-

```
#define LARGE_ITER 100000
#define MED_ITER 1000
#define SEQ_LOOP_ITER 1

for(int i = 0; i<LARGE_ITER; ++i) {
    parlay::parallel_for(0, MED_ITER, [&](size_t j) {
        double acc = 0;
        double prod = 0;
        for(int k = 0; k<SEQ_LOOP_ITER; ++k) {
            prod = (i+j+1)*k;
            acc = acc + prod;
        }
        return acc;
    }, 250);
}
```

Figure 4.7: Synthetic benchmark with low parallelism. The parallel region is designed such that there is a significant amount of idle time for the threads due to load imbalance.

load, keeping non-working durations relatively short and capping energy savings at 11%. In contrast, the sequential merge phase in our `seq_merge` variant forces workers into protracted idle periods, especially at higher thread counts. This behavior creates an ideal environment for elastic scheduling, confirming that putting blocked workers to sleep during severe sequential bottlenecks can yield exceptional energy savings (up to 229%).

4.2.3.6 Synthetic Benchmark with Low Parallelism

We evaluated a synthetically created benchmark specifically designed to exhibit low parallelism. Figure 4.7 presents the code stub for this benchmark. The workload consists of an outer sequential loop executing a large number of iterations. Within each iteration of this outer loop, an inner parallel `for` loop executes with a fixed grain size of 250 iterations. As a result, the inner loop spawns exactly 4 parallel tasks per outer iteration. This structure limits the benchmark to saturating a maximum of 4 threads, leading to inherently low parallelism, particularly at higher thread counts.

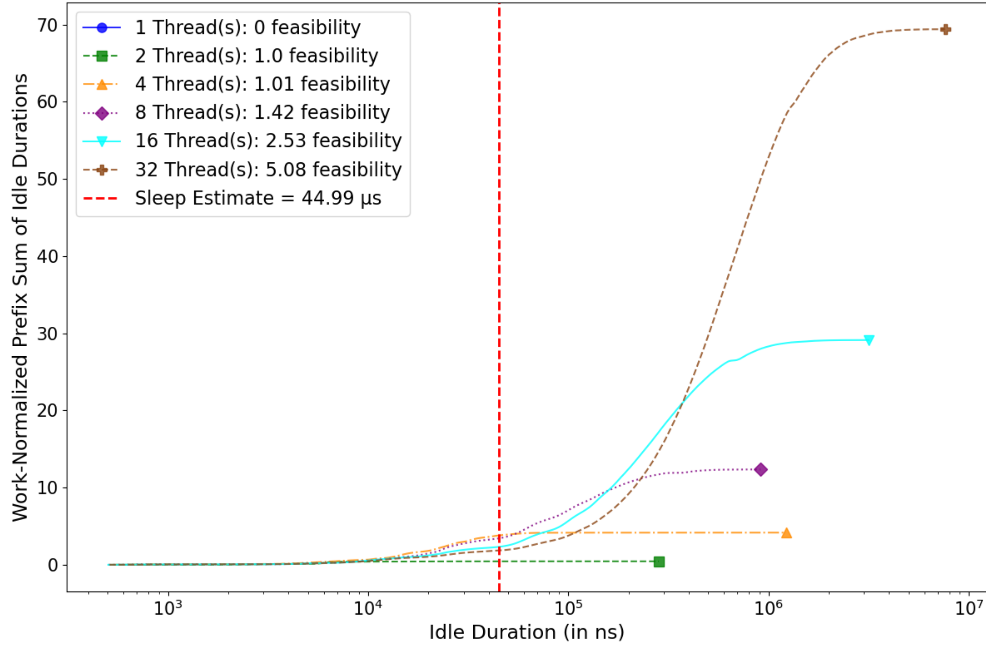


Figure 4.8: Prefix Sum Plot for the synthetic low-parallelism benchmark.

The Prefix Sum Plot in Figure 4.8 confirms this expected behavior. For thread counts within the saturation limit (1, 2, and 4), the Feasibility metric remains close to 1.0, demonstrating negligible potential for energy savings. However, for thread counts exceeding this limit (8, 16, and 32), the metric increases substantially, revealing potential energy savings of approximately 41%, 153%, and 400%, respectively. Furthermore, the theoretical maximum Feasibility—calculated as 71 (1 + 70) under the idealized assumption that TI = FI is exceptionally high compared to the practical feasibility values observed. This discrepancy indicates that while the total accumulated idle time is immense, a majority of the individual idle durations are actually shorter than the estimated sleep threshold. Overall, these results demonstrate that for thread counts $p > 4$, the non-working durations become disproportionately long compared to active working time, exposing a significant opportunity for elastic scheduling to achieve energy savings of up to 400%.

```
#define LARGE_ITER 100000
#define MED_ITER 1000
#define SEQ_LOOP_ITER 100

parlay::parallel_for(0, MED_ITER, [&](size_t i) {
    double acc = 0;
    double prod = 0;
    for(int j = 0; j<LARGE_ITER; ++j) {
        for(int k = 0; k<SEQ_LOOP_ITER; ++k) {
            prod = (i+j+1)*k;
            acc = acc + prod;
        }
    }
    return acc;
});
```

Figure 4.9: Synthetic benchmark with high parallelism. The parallel region is designed such that there is a significant amount of work for the threads to perform, with minimal idle time due to load imbalance.

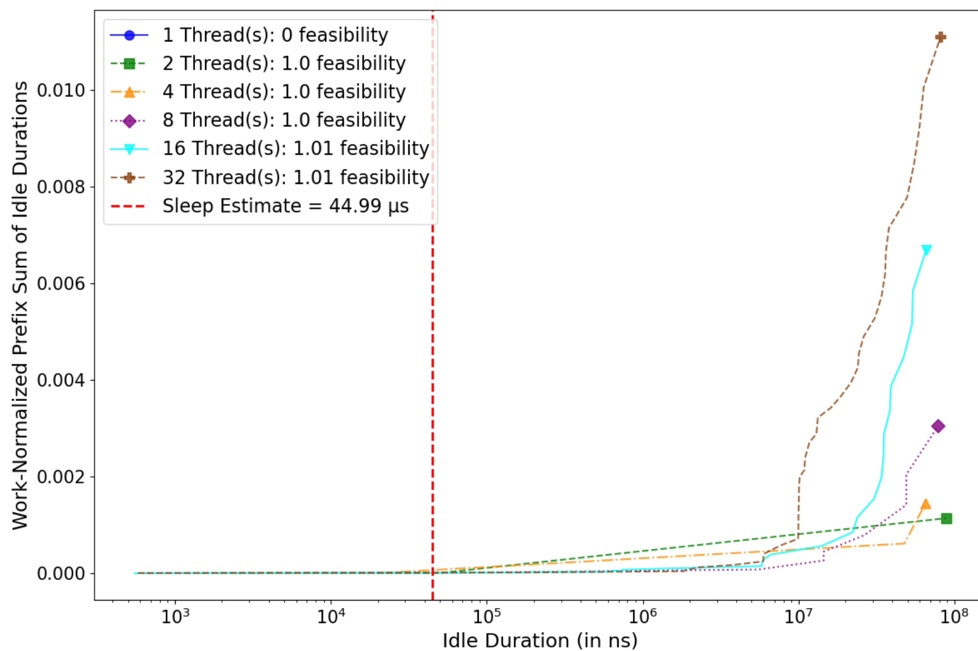


Figure 4.10: Prefix Sum Plot for the synthetic high-parallelism benchmark.

4.2.3.7 Synthetic Benchmark with High Parallelism

To contrast the previous experiment, we evaluated a second synthetic benchmark designed to exhibit exceptionally high parallelism. Figure 4.9 shows the code stub for this workload. The benchmark features an outer parallel `for` loop with 1,000 iterations. Each parallel iteration encapsulates a large sequential loop (100,000 iterations), which in turn contains a nested sequential loop (100 iterations) performing basic multiplication and addition operations. Consequently, this benchmark immediately forks 1,000 parallel tasks, each containing a substantial and identical computational workload. This structure ensures that all available threads are heavily saturated and that work is perfectly balanced without significant stragglers.

The Prefix Sum Plot in Figure 4.10 corroborates this design. The Feasibility metric remains close to 1.0 across all evaluated thread counts, indicating negligible potential for energy savings. Furthermore, the theoretical maximum Feasibility, calculated at a mere 1.014 ($1 + 0.014$), confirms that total idle durations are practically non-existent. Overall, these results illustrate that for highly parallel, perfectly balanced workloads, non-working durations are trivially short compared to working time, rendering sleep-state transitions largely ineffective for reducing energy consumption.

4.2.3.8 Conclusion

In summary, we have proposed a feasibility metric and an accompanying visual representation—the Prefix Sum Plot to quantify the theoretical potential for energy savings based on the distribution of idle durations and their relation to the estimated overhead of putting a processor to sleep. Our feasibility study reveals that benchmarks such as `bfs` and `primes` offer very low potential for energy savings while benchmarks such as `par_mergesort`, `delaunay`, and `seq_merge` have moderate to high potential. Further, we present two synthetic benchmarks specifically designed to exhibit certain feasibility behaviors and verify that our feasibility study through the proposed

metric and plot corroborate with the expectations.

4.2.4 Connecting Feasibility Results to actual Elasticity implementation in ParlayLib

In this section, we connect the theoretical results from our feasibility study to the practical implementation of elasticity in ParlayLib. The elastic scheduler in ParlayLib operates by transitioning workers between three distinct states: *working*, *stealing*, and *sleeping*. At initialization, a single worker begins in the working state while all others are placed in the sleeping state. The scheduler transitions a worker from sleeping to working state in two cases: (1) when an active worker executes a `pardo` construct, spawns new tasks into its local deque, and subsequently wakes a sleeping worker if its deque was previously empty; and (2) when a stealing worker successfully acquires work from a victim's deque and observes that the victim's deque remains non-empty, prompting it to wake another worker. Conversely, if a worker in the stealing state fails to find work after a steal timeout (defined as 10 ms by default), it transitions to the sleeping state. Through this mechanism, the elastic scheduler attempts to dynamically manage the active worker pool—ensuring sufficient threads are awake to process available work while suspending idle workers to conserve energy. The feasibility study quantifies the potential energy savings achievable during these non-working durations, providing a baseline to evaluate how effectively ParlayLib's scheduler converts idle time into energy-saving sleep states without severely degrading end-to-end performance.

We instrumented the scheduler to record the aggregate working, stealing, and sleeping times across all workers, both with and without the elasticity feature enabled. Our primary expectation was that enabling elasticity would yield a substantial increase in total sleeping time for benchmarks exhibiting high feasibility metrics (indicating significant potential for energy savings).

Benchmark	Theoretical Feasibility	Practical Feasibility	Baseline Time: Elasticity OFF (ms)			Δ with Elasticity ON (ms)		
			Working	Stealing	Sleeping	Working	Stealing	Sleeping
bfs	1.04	1.01	173,068.59	7,506.74	145.82	-2,448.31 (0.99x)	-527.68 (0.93x)	+1,612.67 (12.06x)
primes	1.04	1.01	19,683.84	1,138.24	8.38	+776.65 (1.04x)	+74.91 (1.07x)	+96.64 (12.53x)
delaunay	1.46	1.00	561,851.04	370,769.65	1,616.04	-6,567.35 (0.99x)	-9,628.77 (0.97x)	+91.05 (1.06x)
par_mergesort	1.11	1.00	91,949.57	13,519.90	11.11	+1,752.79 (1.02x)	-159.25 (0.99x)	+217.03 (20.53x)
seq_merge	3.29	3.00	69,004.22	155,718.79	7,714.68	+1,418.44 (1.02x)	-147,496.14 (0.05x)	+132,780.66 (18.21x)
synthetic_low_par	5.01	1.07	10,343.43	138,358.22	606.83	+305.53 (1.03x)	+4,967.48 (1.04x)	+109.04 (1.18x)
synthetic_high_par	1.01	1.03	86,194.99	3,810.58	179.81	-79.25 (1.00x)	-3,236.57 (0.15x)	+2,364.51 (14.15x)

Table 4.1: Total working, stealing, and sleeping time (in milliseconds) with elasticity OFF, and the raw performance delta (along with proportional ratio) when elasticity is ON for 32 threads. The Theoretical Feasibility values for each benchmark correspond to the computed values for 32 threads as discussed in 4.2.3, while Practical Feasibility represents $1 + (T_{\text{total_sleep}}/T_{\text{total_work}})$ with elasticity ON for 32 threads.

Table 4.1 presents the absolute baseline times and the performance delta (relative to elasticity OFF) for 32 threads, alongside the computed feasibility values for each benchmark. As expected, enabling elasticity generally increases the total sleeping time across all benchmarks, a direct consequence of the scheduler actively suspending idle threads.

To rigorously evaluate the scheduler's effectiveness, we introduce a *Practical Feasibility* metric, defined as $1 + (T_{\text{sleep}}/T_{\text{work}})$, where T_{sleep} and T_{work} represent the total sleeping time and working time, respectively, when elasticity is ON. This follows from our previous discussion regarding the theoretical maximum of feasibility when $FI = TI$ in 4.2.2.1. By comparing this empirical metric with the theoretical Feasibility value computed in Section 4.2.3, we can determine if the scheduler is optimally exploiting the available energy savings: if the theoretical feasibility and the practical feasibility are of similar orders of magnitude, it denotes that the scheduler is exhibiting the expected behavior with respect to elasticity.

For the `bfs` and `primes` benchmarks, the theoretical Feasibility is low (1.04), indicating minimal potential for energy savings. This theoretical prediction is corroborated by our empirical data: both achieve a Practical Feasibility of just 1.01. The Δ sleeping time for `bfs` (+1,612 ms) and `primes` (+96 ms) is negligible compared to their working times of 173,068 ms and 19,683 ms, respectively. Similarly, the `synthetic_high_par` benchmark exhibits a low theoretical Feasibility of 1.01 and a correspondingly low Practical Feasibility of 1.03.

In contrast, the `seq_merge` benchmark demonstrates a high theoretical Feasibility of 3.29. The data strongly reflects this potential being realized: the scheduler achieves an outstanding Practical Feasibility of 3.00. It successfully converts idleness into 132,780 ms of additional sleeping time, which is highly significant compared to its 69,004 ms working time.

Conversely, `par_mergesort` illustrates a failure to realize even moderate theoretical potential. Despite possessing a theoretical Feasibility of 1.11 (implying an 11% potential for energy savings relative to its working time), it does not achieve any Practical Feasibility (1.00). The scheduler manages a minimal +217 ms increase in sleep time against a massive 91,949 ms baseline working time, indicating a complete inability to capture and convert its available idle phases into sleep.

The `delaunay` benchmark presents an even more significant outlier in this regard. Despite a theoretical Feasibility metric of 1.46 (indicating a potential for roughly 46% energy savings relative to working time), the scheduler fails to convert this idle time into sleep, also yielding a Practical Feasibility of 1.00. The Δ sleeping time is merely +91 ms, virtually zero when compared to the 561,851 ms working time. This highlights a critical shortcoming in ParlayLib's elastic scheduler: it struggles to effectively suspend workers during non-working durations for this specific workload. This inefficiency likely stems from the nature of the sleeping $\rightarrow$ working and stealing $\rightarrow$ sleeping state transitions, which may be too coarse to capture the specific pattern of idle phases in `delaunay`. Another contributing factor could be the underlying implementation of the thread wake-up mechanism. As discussed in Section 4.3.4.1, invoking `atomic_notify_one` during the wake-up routine falls back to `atomic_notify_all` in ParlayLib's default implementation, inadvertently waking all sleeping workers and destroying accumulated sleep time.

The `synthetic_low_par` benchmark exposes this same shortcoming most aggressively. With a remarkably high theoretical Feasibility of 5.01, it possesses enormous potential for energy savings (further evidenced by its massive 138,358 ms baseline stealing time). Yet, the scheduler achieves a meagre Practical Feasibility of just 1.07, managing to increase sleep time by a negligible +109 ms. Despite having approximately 400% potential energy savings available in idle time, the scheduler

does not capitalize on it. The reasons for this can be speculated to be similar to the ones discussed above - coarse state transitions and the wake-up all sleeping threads mechanism.

In summary, our evaluation demonstrates that while the proposed Feasibility metric serves as a strong theoretical indicator for potential energy savings, the current implementation of ParlayLib's elastic scheduler exhibits mixed success in realizing this potential. For workloads like `seq_merge`, the scheduler effectively bridges theory and practice, successfully converting massive amounts of idleness into energy-saving sleep states. This demonstrates the effectiveness of the ParlayLib scheduler for coarse transitions between fully parallel and fully sequential behavior or vice-versa. Elasticity in ParlayLib seems to be designed with this as one of the driving factors. However, the glaring discrepancies between theoretical and practical Feasibility observed in `par_mergesort`, `delaunay`, and `synthetic_low_par` highlight the shortcomings of the scheduler. The failure to capitalize on moderate to massive theoretical idle time points directly to inefficiencies in the scheduler's coarse-grained state transitions—and the costly mass wake-up call caused by the `atomic_notify_all` fallback.

4.3 Race Condition in the Elasticity Implementation in ParlayLib

During our evaluation of the existing elasticity implementation in `ParlayLib`, we identified a buggy race condition that occurs during the scheduler's initialization phase. This section details our analysis of the race condition, explores its root cause, and presents our proposed solution to resolve it. Additionally, we identified a potential deadlock scenario stemming from the operation ordering within the `wait_for_work` function. We outline this deadlock scenario, propose a corrected operation ordering, and provide an argument that our proposed ordering successfully prevents the deadlock.

Figure 4.11 presents the relevant code snippets from `ParlayLib`, specifically the original

implementations of the wait_for_work and wake_up_a_worker functions involved in the race condition. For the complete scheduler implementation, readers are referred to the /include/parlay/scheduler.h file within the ParlayLib codebase.

```
// Wait until notified to wake up
void wait_for_work() {
    num_awake_workers.fetch_sub(1);
    parlay::atomic_wait(&wake_up_counter, wake_up_counter.load());
    num_awake_workers.fetch_add(1);
}

// Wakes up at least one sleeping worker (more than one
// worker may be woken up depending on the implementation).
void wake_up_a_worker() {
    if (num_awake_workers.load(std::memory_order_acquire) < num_threads) {
        wake_up_counter.fetch_add(1);
        parlay::atomic_notify_one(&wake_up_counter);
    }
}
```

Figure 4.11: Original code for wait_for_work and wake_up_a_worker functions.

4.3.1 Reproducing the Race Condition

To illustrate how this race condition can be triggered, Figure 4.12 provides the code for a minimal reproducible example. It considers an execution scenario with two worker threads. The race condition unfolds through the following sequence of events:

1. At the exact moment of scheduler initialization, the internal state variable num_awake_workers is initialized to 2.

2. Immediately after Thread 0 (the primary worker) finishes initializing the scheduler, Thread 1 is logically marked as "awake." However, it may not be scheduled for execution by the operating system immediately due to standard context-switching delays.

3. Suppose Thread 0 now proceeds to execute a pardo construct, which internally invokes scheduler.spawn(&right_job). Following this, it evaluates the condition if (first)

wake_up_a_worker(). However, this function does not execute atomic_notify_one() because it perceives the system to be fully awake (the current num_awake_workers is 2, which equals num_threads).

4. Subsequently, Thread 1 is finally scheduled by the OS and begins execution. It enters the worker() routine, finds no immediate jobs in its deque, and proceeds to wait_for_work(). Because it perceives no available work, it immediately transitions to a sleep state, blocking on the wake_up_counter synchronization variable.

This sequence results in an inefficiency: Thread 0 holds a spare job in its local deque, but Thread 1 is asleep and entirely unaware of the available work, preventing it from stealing and executing the task.

```
#define NUM_ITER 1000000000

double seq_loop(double num) {
    double acc = 0, prod = 0;
    for(int i = 0; i<NUM_ITER; ++i) {
        prod = num*i;
        acc = acc + prod;
    }
    return acc;
}

int main(int argc, char* argv[]) {
    double ans1 = seq_loop(atoi(argv[1]));
    double ans2 = 0, ans3 = 0;

    parlay::par_do(
      [&]() { ans2 = seq_loop(ans1); },
      [&]() { ans3 = seq_loop(ans1 + atoi(argv[1])); }
    );

    double ans4 = seq_loop(ans2 + ans3);
    cout << ans4 << "\n";

    return 0;
}
```

Figure 4.12: Example code that triggers race condition

4.3.2 Proposed Solution

```
void wait_for_work() {
    auto orig_val = wake_up_counter.load();
    num_awake_workers.fetch_sub(1);

    size_t id = worker_id();
    Job* job = nullptr;
    for (size_t i = 0; i <= YIELD_FACTOR * num_deques; i++) {
      if (finished()) {
        num_awake_workers.fetch_add(1);
        return;
      }
      job = try_steal(id);
      if(job) {
        num_awake_workers.fetch_add(1);
        (*job)();
        return;
      }
    }

    parlay::atomic_wait(&wake_up_counter, orig_val);
    num_awake_workers.fetch_add(1);
}
```

Figure 4.13: Updated code for `wait_for_work` function that resolves the race condition with the changes highlighted.

We propose the following solution to resolve the aforementioned race condition:

1. Within the `wait_for_work` routine, first announce the intent to transition the current worker to sleep state by decrementing the `num_awake_workers` counter.
2. However, before committing to the sleep state via `atomic_wait`, perform one final round of steal attempts:
   (a) If a job is successfully stolen: immediately increment `num_awake_workers` to reflect the active state, and begin executing the job, bypassing the `atomic_wait` entirely.
   (b) If no job is found during this final check: safely proceed to execute `atomic_wait`.

Figure 4.13 illustrates the modified code snippet for the `wait_for_work` function, highlighting the changes made to implement this solution. By introducing this additional check for available work immediately before sleeping, we ensure that no worker transitions to sleep state while there is still executable work available, effectively eliminating the race condition.

4.3.3 Execution Order of Loading `wake_up_counter` and Decrementing `num_awake_workers`

While implementing the proposed solution within the `wait_for_work` function, we observed that the execution order between loading `wake_up_counter` and decrementing `num_awake_workers` impacts the correctness of the scheduling logic. The current implementation in `ParlayLib` first decrements `num_awake_workers` and then loads the `wake_up_counter`. However, our analysis demonstrates that the inverse order—loading the `wake_up_counter` prior to decrementing `num_awake_workers`—is necessary to ensure correctness.

Only three functions modify or access `wake_up_counter` and `num_awake_workers`: `wake_up_a_worker`, `wake_up_all_workers`, and `wait_for_work`. For the purpose of this analysis, `wake_up_a_worker` and `wake_up_all_workers` exhibit equivalent behavior. The necessity of our proposed ordering becomes particularly evident in a system constrained to two worker threads. While a higher thread count or subsequent task spawns may eventually allow the scheduler to self-correct regardless of the ordering, a two-thread scenario reliably isolates the bug.

We consider the following scenarios to illustrate the necessity of the proposed ordering:

4.3.3.1 Example Scenario 1 (Ex1): Original Ordering

This scenario evaluates the original execution order: (1) decrement `num_awake_workers`, followed by (2) load `wake_up_counter`. We assume the initial state `num_awake_workers =`

`num_threads` = 2.

Consider an interleaving where worker `w1` executes `wait_for_work` concurrently while worker `w2` executes `wake_up_a_worker`.

Step	`wait_for_work` (worker `w1`)	`wake_up_a_worker` (worker `w2`)
1	`num_awake_workers.fetch_sub(1);`	
2		`if (num_awake_workers < num_threads)` `wake_up_counter.fetch_add(1);`

3	`orig_val = wake_up_counter.load();`
	`parlay::atomic_wait(`
	`&wake_up_counter, orig_val);`	

4		`parlay::atomic_notify_one(` `&wake_up_counter);`

Table 4.2: Execution interleaving for the Ex1 Scenario (Original Ordering).

The execution interleaving detailed in Table 4.2 illustrates a sequence where the `atomic_wait` instruction on worker `w1` blocks indefinitely, even after the notification call from worker `w2`. This occurs because, at the moment of evaluation, the current value of `wake_up_counter` matches the stale value previously loaded into `orig_val`.

4.3.3.2 Example Scenario 2 (Ex2): Proposed Ordering

This scenario evaluates our proposed execution order: (1) load `wake_up_counter`, followed by (2) decrement `num_awake_workers`. Again, we assume the initial state `num_awake_workers` = `num_threads` = 2.

The interleaving detailed in Table 4.3 demonstrates a sequence where the `atomic_wait` instruction on worker `w1` successfully terminates following the notification call from worker `w2`. This correct behavior is achieved because the current value of `wake_up_counter` differs from the value loaded into `orig_val`, causing the wait condition to fall through.

It should be noted that the three `atomic_wait` statements (labeled 1, 2, and 3 in the table) denote three distinct possible execution interleavings for `w1`. Under our proposed ordering, all three

Step	`wait_for_work` (worker w1)	`wake_up_a_worker` (worker w2)
1	`orig_val = wake_up_counter.load();` `num_awake_workers.fetch_sub(1);` `parlay::atomic_wait(` `&wake_up_counter, orig_val); //1`	
2		`if (num_awake_workers.load(acq)` `< num_threads) //true`
3	`parlay::atomic_wait(` `&wake_up_counter, orig_val); //2`	
4		`wake_up_counter.fetch_add(1);` `parlay::atomic_notify_one(` `&wake_up_counter);`
5	`parlay::atomic_wait(` `&wake_up_counter, orig_val); //3`	

Table 4.3: Execution interleaving for the Ex2 Scenario (Proposed Ordering).

interleavings correctly result in the termination of the wait state on `w1`.

4.3.4 Correctness of the load-before-decrement order

In this section, we argue that the proposed load-before-decrement ordering is correct, in the sense that it guarantees forward progress of the elastic scheduler. To achieve this, we synthetically construct a counter-example designed to fail—similar to the Ex1 scenario—where a worker loads the `wake_up_counter` *after* another worker has already updated it.

Step	`wait_for_work` (worker w1)	`wake_up_a_worker` (worker w2)	`wait_for_work` (worker wx)
1			`orig_val = wake_up_counter.load();` `num_awake_workers.fetch_sub(1);` `parlay::atomic_wait(` `&wake_up_counter, orig_val);`
2		`if (num_awake_workers.load(acq)` `< num_threads) //true` `wake_up_counter.fetch_add(1);`	
3	`orig_val = wake_up_counter.load();` `num_awake_workers.fetch_sub(1);` `parlay::atomic_wait(` `&wake_up_counter, orig_val);`		
4		`parlay::atomic_notify_one(` `&wake_up_counter);`	

Table 4.4: Execution interleaving for the Informal Proof Scenario.

Table 4.4 captures the exact interleaving scenario used for the proof by contradiction. Consider

a scenario involving multiple workers. Let worker w1 execute wait_for_work and worker w2 execute wake_up_a_worker. For the wake_up_counter update in w2 to execute, its internal if condition (num_awake_workers < num_threads) must evaluate to true.

However, at this exact moment, w1 has not yet decremented num_awake_workers because it has not yet loaded the wake_up_counter. Therefore, for w2's if condition to be true, there *must* exist at least one other worker (which we will denote as wx) that is already asleep (i.e., it has previously decremented num_awake_workers and entered a wait state).

Assume the following worst-case interleaving:

1. w2 successfully evaluates the if condition and increments the wake_up_counter.
2. w1 loads the newly updated wake_up_counter, decrements num_awake_workers, and enters the atomic_wait instruction.
3. w2 executes the atomic_notify_one call.

According to standard C++ semantics, if multiple threads are blocked in an atomic waiting operation on the same object, notify_one unblocks *at least one* of those threads. Crucially, the standard provides no guarantee regarding *which* specific thread will be unblocked; this is entirely implementation-defined.

We analyze how this resolves under different underlying implementations:

4.3.4.1 ParlayLib's Default Behavior (atomic_notify_all fallback)

In most platform versions implemented within ParlayLib (include/parlay/internal/atomic_wait.h), atomic_notify_one is internally implemented as a call to atomic_notify_all. In this scenario, the notification from w2 will unblock *both* w1 and wx.

When w1 unblocks, it evaluates its wait condition. Since the current value of the wake_up_counter equals w1's loaded value, w1 will resume waiting. However, wx is also un-

blocked. Because `wx` successfully went to sleep earlier, we are guaranteed that its loaded value of the `wake_up_counter` predates `w2`'s recent update. Therefore, for `wx`, the current counter value differs from its originally loaded value. This discrepancy terminates its wait, allowing `wx` to wake up and make forward progress.

4.3.4.2 Strict `notify_one` Implementations (e.g., Win32)

In specific environments like Win32, the implementation strictly notifies exactly one thread using the following system calls:

- `WaitOnAddress`: Waits for the value at the specified address to change.
- `WakeByAddressSingle`: Wakes a single waiting thread. If multiple threads are waiting, the system wakes the *first* thread that went to sleep.

Because `wx` decremented `num_awake_workers` and went to sleep before `w1` even began its wait process, `wx` is guaranteed to be the "first" thread in the queue. Consequently, `WakeByAddressSingle` will specifically notify and wake `wx`. Once again, `wx` will observe the changed counter, terminate its wait, and make forward progress.

4.3.4.3 Conclusion

In all valid implementations, even if `w1` improperly blocks due to the worst-case interleaving, the pre-existing sleeping thread `wx` is guaranteed to be woken up in its place. Because the scheduler only requires that *a* worker wakes up to handle the available work, the system consistently makes forward progress. This demonstrates that the load-before-decrement ordering is logically correct and prevents deadlock.

Caveat: While this informal proof holds, it relies heavily on the underlying `atomic_notify_one` implementation guarantees. A trivial, implementation-agnostic way to guarantee this

safe behavior universally would be to explicitly replace the `atomic_notify_one` call with `atomic_notify_all`.

4.3.5 Verification of the Fix

To evaluate the correctness of our proposed solution, we implemented the modified `wait_for_work` shown in Figure 4.13 within a local branch of `ParlayLib`. We first executed the minimal reproducible example (detailed in Figure 4.12) that previously triggered the original race condition. Under the patched implementation, the previously hanging thread successfully awakens and processes the available tasks, confirming the immediate resolution of the race. To validate the robustness of the fix, we subsequently stress-tested the modified scheduler across a variety of workloads and thread configurations. Throughout these tests, the system consistently exhibited correct execution without any instances of hangs or deadlocks. We must note however, that we were not able to create a test case that succesfully triggered the deadlock due to the original decrement-before-load ordering. Hence, we could not empirically verify the correctness of the proposed load-before-decrement ordering.

Further, we executed the comprehensive suite of `ParlayLib` benchmarks to ensure the absence of performance or correctness regressions. The empirical results indicated that the proposed fix introduces no unintended side effects. Notably, the patched scheduler actually yielded an average improvement of 2.0% in wall-clock time and 1.9% in CPU time across the benchmark suite. This modest performance gain may be attributable to the elimination of the race condition and the corresponding reduction in spurious thread stalling.

5 Towards a definition of Fluctuation

5.1 Motivation

Ideally, an elastic scheduler should guarantee that workloads exhibiting high theoretical feasibility (as defined in Section 4.2) yield commensurately high practical energy savings and practical feasibility (as defined in Section 4.2.3). However, our empirical analysis of the elastic scheduler in ParlayLib (Section 4.2.3) reveals that this correlation does not always hold. While the scheduler successfully translates idleness into energy-saving sleep states for workloads like `seq_merge`, benchmarks such as `par_mergesort`, `delaunay`, and `synthetic_low_par` exhibit a stark gap between their theoretical feasibility and the practical energy reduction achieved through elasticity. Motivated by these discrepancies, we pose the following research question: *Can we capture and characterize a structural feature of a computation DAG that directly correlates with the scheduler's ability to extract energy savings?* Specifically, we aim to define an analytical metric—which we term ***fluctuation***—capable of classifying DAGs based on their intrinsic suitability for elastic sleep transitions.

While the DAG-based work-span model is simple and effective, it does not capture such finer-grained characteristics as variations in how work is distributed throughout the execution. Two computations may exhibit the exact same asymptotic work and span measures, yet differ signifi-

cantly in their structural shape and behavior.

(a) Infrequent synchronization (low fluctuation).

(b) Frequent synchronization (high fluctuation).

Figure 5.1: Two cases of DAG structures under a strict binary fork-join model.

Consider the two extreme computational structures illustrated in Figure 5.1. Figure 5.1(a) depicts a computation characterized by a single, initial fork that spawns two independent branches. Each branch executes a long, continuous sequence of operations before synchronizing at a final join at the end. Such a structure may induce considerable processor idleness if the workloads of the two branches are uneven leading to stragglers. These extended idle phases present opportunities for energy savings, as explored in Section 4.2.

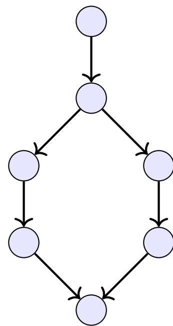
In contrast, Figure 5.1(b) illustrates a computation characterized by a continuous chain of highly granular forks and immediate joins. If the tasks corresponding to a particular fork are identical in terms of their work, this structure may not experience significant processor idleness. Consequently, this continuous cycle renders energy-saving sleep states infeasible due to the absence of idle durations and places a heavy, persistent burden on the scheduler.

Therefore, there is a clear need for more nuanced cost models that can capture these structural differences beyond just total work and span. Such models would enable a deeper understanding of parallel computations and inform the design of more efficient algorithms and schedulers. This

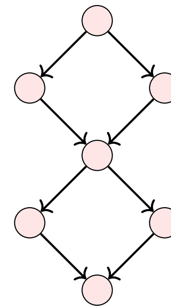
motivates our exploration into defining a metric of *fluctuation* that specifically quantifies the variability in the distribution of work across the execution of a computation graph.

5.2 Equivalent Work and Span Measures

As an initial step towards defining a metric of fluctuation, we consider two DAGs that share the same asymptotic work and span but differ structurally.



(a) DAG with a single fork ($F = 1$).



(b) DAG with frequent forks ($F = 2$).

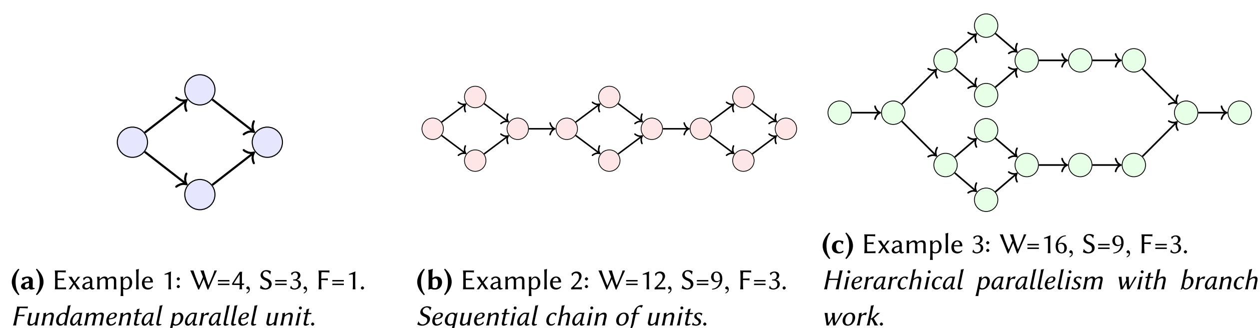
Figure 5.2: Two computational structures with identical total work ($W = 7$) and span ($S = 5$), but varying degrees of structural fluctuation dictated by the number of forks (F).

For instance, consider the two DAGs illustrated in Figure 5.2. Figure 5.2(a) depicts a DAG with a single initial fork, followed by a sequence of independent operations before joining at the end. In contrast, Figure 5.2(b) depicts a DAG with frequent forks and joins throughout its execution. Assuming every node in the DAG represents a unit of work, while both DAGs possess exactly identical total work ($W = 7$) and a critical path span ($S = 5$), they exhibit different patterns of parallelism and processor utilization. For these two computations to differ in structure while maintaining the same work and span, they must differ in their distribution of work across the execution timeline. This difference is directly linked to the total number of forks, and thus the number of synchronization points. This suggests that the total number of forks, F (or equivalently, the number of joins), is a necessary baseline component for modeling fluctuation.

5.3 Differing Work and Span Measures

Now, we consider the more general case of DAGs that may differ in their work and span measures. In order to model fluctuation in this case, the intuition is that we need to consider additional metrics beyond just the number of forks. A starting point could be to compute the F/S ratio for each DAG, which attempts to capture the average number of forks over the span of the computation. However, this ratio alone may not be sufficient to capture the full extent of fluctuation, as it does not account for how work is distributed across the execution.

For instance, consider the examples illustrated in Figure 5.3. Example 1 represents a basic parallel task with $W = 4$, $S = 3$, and $F = 1$. Example 2 represents a sequential chain of three such tasks, resulting in $W = 12$, $S = 9$, and $F = 3$. Example 3 introduces a more complex hierarchical structure with nested parallelism and additional sequential work within its branches, yielding $W = 16$, $S = 9$, and $F = 3$. Despite their significantly different structures and varying distributions of work, all three DAGs exhibit the exact same F/S ratio of $1/3$. Because they share this identical ratio, relying solely on F/S would falsely suggest they impose the same relative fluctuation, leading to identical estimates for scheduling burden and synchronization overhead.



(a) Example 1: W=4, S=3, F=1. *Fundamental parallel unit.*

(b) Example 2: W=12, S=9, F=3. *Sequential chain of units.*

(c) Example 3: W=16, S=9, F=3. *Hierarchical parallelism with branch work.*

Figure 5.3: Three diverse general DAG cases demonstrating that distinct computational structures with differing work (W) and span (S) can possess identical F/S ratios (1/3 for all), revealing the ratio's insufficiency to model fluctuation. However, the $F/(W \times S)$ values are distinct, 1/12, 1/36, and 1/48 respectively.

Therefore, the F/S ratio is not sufficient to model fluctuation, and we need to consider additional metrics. A key observation is that the F/S ratio does not account for the total work performed

within the DAG. Therefore, a more comprehensive metric of fluctuation could be defined as F/S normalized by the total work, which captures both the frequency of forks and the distribution of work across the execution.

This could be expressed as $F/(W \times S)$, where W is the total work of the DAG. This metric provides a more nuanced measure of fluctuation that accounts for both the structure of the DAG and the amount of work being performed. By incorporating total work into the denominator, the new $F/(W \times S)$ ratio effectively differentiates the DAGs in the examples. Specifically, it yields $1/12$ for Example 1, $3/108$ (simplifying to $1/36$) for Example 2, and $3/144$ (simplifying to $1/48$) for Example 3. This progressive decrease successfully captures the differences in their structure and behavior—demonstrating that as more work is packed into the computation relative to its synchronization points, the relative scheduling burden drops—which the original F/S ratio entirely failed to distinguish.

5.4 Shortcomings and Future Work

While the proposed $F/(W \times S)$ ratio provides a considerably more nuanced measure of fluctuation than the unnormalized F/S metric, it retains several theoretical and practical limitations.

Firstly, as an aggregated global metric, it may fail to capture local variability in the work distribution. Because it averages the density of forks across the entire execution timeline, a computation with evenly distributed parallelism could yield the exact same ratio as a computation featuring a massive, dense burst of parallelism followed by a lengthy sequential phase. Furthermore, structural metrics abstract away critical hardware realities; the ratio does not account for external factors that heavily influence actual scheduling burden, such as variable communication costs, inter-thread load imbalances, and contention for shared memory resources.

Secondly, the $F/(W \times S)$ ratio currently functions solely as a *relative* comparative tool rather than an absolute measure. While it successfully ranks different DAGs by their structural schedul-

ing burden, the raw numerical value itself lacks a definitive, standalone interpretation. There is currently no mathematically defined threshold indicating what specific value constitutes "high" versus "low" fluctuation; this threshold is highly likely to vary depending on the specific context of the algorithm and the target hardware architecture. This was precisely our observation from our attempts to empirically generate the fluctuation metric for various benchmarks. Using our previously proposed scheduler augmentation framework, we were able to design a vertex to track Work, Span and number of Forks for various benchmarks. Using these generated values, we trivially computed the $F/(W \times S)$ ratio for these benchmarks. We observed that metric provided no absolute meaning and was moderately successful in acting as a comparative tool to rank the benchmarks based on their suitability for energy savings.

Ultimately, while the introduction of the $F/(W \times S)$ ratio represents a step toward quantifying structural fluctuation, it is not a complete model. Further research is required to refine this metric.

6 | Related Work

The research presented in this thesis builds upon decades of foundational work in parallel computing, specifically focusing on task scheduling, energy efficiency, performance modeling, and execution profiling. This chapter contextualizes our contributions by reviewing the existing literature across these primary domains.

6.1 Parallel Programming Models

In this thesis, we consider the binary fork-join model as our primary programming model. However, we note there exist other models as detailed below.

Futures and Promises. Unlike the strict synchronization of fork-join, the *futures* (or promises) model allows developers to spawn a task that immediately returns a proxy object representing a value that will become available later [5]. A thread only blocks when it explicitly attempts to read the value of an unresolved future. This allows for the construction of arbitrary, non-series-parallel dependencies.

Async/Await and Coroutines. Originating from the need to handle highly concurrent, I/O-bound operations, the *async/await* model relies on stackless coroutines or lightweight user-level threads. When a task awaits a blocking operation, it yields control back to the runtime, allowing the underlying OS thread to execute other ready tasks. While extremely popular in languages

like Go (goroutines), Rust, and Python for managing massive concurrency [6], the async/await paradigm is generally optimized for latency hiding rather than throughput.

6.2 Parallel Languages

Historically, providing high-level parallel abstractions required the development of entirely new languages or heavy compiler extensions. The High Productivity Computing Systems (HPCS) initiative spurred the creation of Partitioned Global Address Space (PGAS) languages like Chapel [7] and X10 [8]. These languages treat parallelism as a first-class citizen, offering rich, domain-specific syntax for task distribution. Similarly, Cilk [9] extended C/C++ with dedicated keywords, relying on specialized compiler passes to generate highly optimized task closures. OpenMP also emerged as an industry standard, utilizing compiler pragmas to inject multi-threading logic into sequential code [10].

6.3 Foundations of Work-Stealing

The work-stealing scheduling paradigm has become the de facto standard for executing dynamic multithreaded computations. Foundational research in this domain [11] proved that randomized work-stealing achieves nearly optimal (within a constant factor of 2) execution time bound for fully strict multithreaded computations.

The ABP deque [12] provided a non-blocking, concurrent double-ended queue implementation that minimized synchronization overhead between workers and stealers. This structure mathematically guaranteed strong complexity bounds for work-stealing schedulers in shared-memory environments. These guarantees underpin almost all modern lightweight scheduling frameworks, validating the base scheduling efficiency before energy or structural fluctuations are even considered.

6.4 Profiling Parallel Programs

One application of scheduler augmentation is as a source-level profiling technique, which has natural limitations in terms of precision and overhead, especially for time-based metrics (see discussion in [13]). Nevertheless, it can be a useful tool for quickly obtaining useful insights about performance. Specialized tools for particular languages can offer more precision, such as CilkView [14] and CilkProf [15]. Both tools perform incremental maintenance of the computation graph by dynamically updating local information at each executed fork/spawn and join/sync, which is similar to our approach. Basso et al. [16] considered profiling in a setting with more expressive fork-join tasks (in particular, features such as task cancellation). There has also been a large body of work on HPCToolkit [13], which offers a powerful tool suite for profiling parallel programs based on sampling and binary analysis. In the context of this prior work, one benefit of scheduler augmentation is that it enables basic graph-based profiling in languages and environments where such tools are not available.

6.5 Scheduler-based Metrics

Collecting performance metrics directly from the scheduler (such as idleness, steal counts, etc.) can be useful for pinpointing particular bottlenecks, especially scheduler overheads and insufficient parallelism [17, 18]. Lifflander, Krishnamoorthy, and Kalé [19] develop a general technique for collecting traces of scheduler events in work-stealing schedulers, which bears some resemblance to our approach, except that they seek to incorporate schedule-specific details into trace, whereas we seek to abstract away the schedule and instead focus on the underlying computation graph.

6.6 Lightweight Scheduling Libraries

Various "library-only" aproaches are available across many languages, such as Intel Threading Building Blocks [1], the Rust Rayon library [20], the Java Fork/Join framework [21], `domainslib` in multicore OCaml [22, 23], etc. The advantage of these libraries is that they require no special compiler support, and can easily be integrated into a larger project, even just to parallelize a single key subroutine. However, as far as we know, no existing lightweight library provides support for incremental observation or maintenance of the computation graph during execution. We believe that scheduler augmentation will be applicable across all of these libraries, with only small modifications to the scheduler implementation, and we are planning to investigate this in future work.

6.7 Cilk Hyperobjects

The implementation of Cilk hyperobjects [24] bears some resemblance to scheduler augmentation: each hyperobject maintains a small amount of state across workers, and this state is combined (monoidally) at join-points, automatically. In some sense, the functionality of hyperobjects can be thought of as a form of scheduler augmentation, by viewing the hyperobject state as vertex-local data and observing that vertex joining rules can be derived from the corresponding monoid of the hyperobject.

6.8 Granularity Control

A well-known challenge in parallel programming is the overhead of task creation. Effectively amortizing the cost of the work-stealing deque requires optimal granularity control. Extensive research has addressed this via oracle-guided execution [25] and dynamic cutoff heuristics that

adapt task sizes based on instantaneous load [26, 27].

6.9 Modern Parallel Benchmarking and Frameworks

Our empirical evaluations and framework implementations heavily leverage modern, state-of-the-art parallel infrastructures. To standardize our evaluations, we utilize the Problem Based Benchmark Suite (PBBS) [3], which provides highly optimized implementations for algorithmic problems, enabling robust comparisons of parallel overheads.

Further, our modifications are built directly into ParlayLib [2], a modern C++ library offering a parallel scheduler and fundamental parallel primitives. Similar library-only approaches to scheduling are prevalent across many programming languages. Because these libraries uniformly rely on standard work-stealing algorithms, we believe our Scheduler Augmentation techniques and elasticity optimizations will be broadly applicable across various runtime environments.

6.10 Energy Efficiency and Waste-Efficient Scheduling

While theoretically optimal for performance, traditional work-stealing algorithms assume a dedicated execution environment where processors constantly poll or spin-wait when idle, leading to energy waste.

A strategy to mitigate waste (defined as unsuccessful steal attempts) is presented in [28], which introduces waste-efficient work stealing as an extension to randomized work-stealing as a strategy to mitigate waste . They show how their framework has no significant performance overhead while reducing and bounding waste. Other approaches such as [29] to elasticity dynamically resize the active thread pool based on system load or parallelism feedback.

6.11 Performance Modeling

Theoretical reasoning about parallel computations heavily relies on the Work-Span (or Work-Depth) DAG model, a concept extensively popularized by the Cilk project [9, 30]. While foundational for proving asymptotic bounds, the vanilla Work-Span model abstracts away the physical costs of thread synchronization, deque operations, and cache-miss penalties.

To capture these practical overheads, researchers introduced the concept of the burdened span [31], which extends the traditional critical path length by incorporating the latency of concurrent interactions. While the burdened span effectively penalizes computations with high synchronization costs on the critical path, it often fails to capture the aggregate scheduling burden distributed across highly parallel, non-critical branches. Our proposed $F/(W \times S)$ fluctuation metric serves as a complementary approach, globally quantifying the density of synchronization to robustly model structural fluctuation.

7 | Conclusion

In this thesis, we have presented techniques to analyze and debug the performance of parallel programs within lightweight scheduling frameworks across three main areas:

- **Granularity Analysis using Scheduler Augmentation.** We have presented scheduler augmentation, a technique that enables the programmer to directly observe the computation graph of a parallel execution by providing vertex-local data definitions to the scheduler. The technique is applicable even in library-only approaches, with no special compiler support. We use scheduler augmentation to address the granularity control problem with a detailed granularity analysis of the Parallel Range Query benchmark. Our technique successfully identified specific subcomputations—particularly within the third phase of the algorithm—that were excessively fine-grained and, hence, prime candidates for algorithmic coarsening.

- **Investigating the Effectiveness of Elasticity.** To evaluate the energy efficiency of elastic work-stealing schedulers, we formulated a theoretical *feasibility metric* and an accompanying visual representation, the *Prefix Sum Plot*. These tools allow us to rigorously quantify the theoretical potential for energy savings by analyzing idle time distributions across a variety of algorithmic benchmarks. We then transitioned from theory to practice, evaluating the effectiveness of the elasticity implementation in ParlayLib by verifying the achieved practical energy savings (measured in realized sleep time) as compared to the theoretical feasibility values. We observe that the elastic implementation in Par-

layLib successfully converts massive amounts of idleness into energy-saving sleep states for workloads like `seq_merge`. However, for benchmarks like `par_mergesort`, `delaunay`, and `synthetic_low_par`, there is a considerable gap between the practically achieved feasibility and the theoretical potential, highlighting the shortcomings of the scheduler. Further, as a derivative of this work, we identified a critical race condition in the ParlayLib scheduler initialization and a possible deadlock scenario. We proposed solutions for these through a "steal-after-sleep-announcement" protocol and a "load-before-decrement" instruction ordering.

- **Towards a definition of Fluctuation.** Given our observation that practical elasticity implementations, such as the one in ParlayLib, exhibit mixed success in converting theoretical potential into practical energy savings, we propose a *fluctuation* metric to answer the research question: *Can we capture and characterize a structural feature of a computation DAG that directly correlates with the scheduler's ability to extract energy savings?* We analyze this metric while noting its significant shortcomings and need for refining and future work.

8 References

[1] UXL Foundation. *oneAPI Threading Building Blocks*. 2025. URL: https://uxlfoundation.github.io/oneTBB/ (visited on 08/01/2025).

[2] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. "ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines". In: *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*. Ed. by Christian Scheideler and Michael Spear. ACM, 2020, pp. 507–509. ISBN: 978-1-4503-6935-0.

[3] Daniel Anderson et al. "The problem-based benchmark suite (PBBS), V2". In: *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*. Ed. by Jaejin Lee, Kunal Agrawal, and Michael F. Spear. ACM, 2022, pp. 445–447. ISBN: 978-1-4503-9204-4.

[4] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. "PAM: parallel augmented maps". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. Ed. by Andreas Krall and Thomas R. Gross. ACM, 2018, pp. 290–304. ISBN: 978-1-4503-4982-6.

[5] Robert H Halstead Jr. "MULTILISP: A language for concurrent symbolic computation". In: *ACM SIGPLAN Notices*. Vol. 20. 8. ACM. 1985, pp. 19–30.

[6] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting coroutines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.2 (2009), pp. 1–31.

[7] Bradford L Chamberlain, David Callahan, and Hans P Zima. "Parallel programmability and the Chapel language". In: *The International Journal of High Performance Computing Applications*. Vol. 21. 3. Sage Publications Sage UK: London, England, 2007, pp. 291–312.

[8] Philippe Charles et al. "X10: an object-oriented approach to non-uniform cluster computing". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 519–538.

[9] Matteo Frigo, Pablo Halbherr, and Charles E Leiserson. "The implementation of the Cilk-5 multithreaded language". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 212–223.

[10] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[11] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[12] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. "Thread scheduling for multiprogrammed multiprocessors". In: *Theory of Computing Systems* 34.2 (2001), pp. 115–144.

[13] Laksono Adhianto et al. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: *Concurr. Comput. Pract. Exp.* 22.6 (2010), pp. 685–701.

[14] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. "The Cilkview scalability analyzer". In: *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*. Ed. by Friedhelm Meyer auf der Heide and Cynthia A. Phillips. ACM, 2010, pp. 145–156. ISBN: 978-1-4503-0079-7.

[15] Tao B. Schardl et al. “The Cilkprof Scalability Profiler”. In: *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*. Ed. by Guy E. Blelloch and Kunal Agrawal. ACM, 2015, pp. 89–100. ISBN: 978-1-4503-3588-1.

[16] Matteo Basso et al. “Accurate Fork-Join Profiling on the Java Virtual Machine”. In: *Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings*. Ed. by José Cano and Phil Trinder. Vol. 13440. Lecture Notes in Computer Science. Springer, 2022, pp. 35–50.

[17] Nathan R. Tallent and John M. Mellor-Crummey. “Identifying Performance Bottlenecks in Work-Stealing Computations”. In: *Computer* 42.11 (2009), pp. 44–50.

[18] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. “Parallel Work Inflation, Memory Effects, and their Empirical Analysis”. In: *CoRR* abs/1709.03767 (2017).

[19] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kalé. “Steal Tree: low-overhead tracing of work stealing schedulers”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 507–518. ISBN: 978-1-4503-2014-6.

[20] The Rust Project Developers. *Rayon*. 2025. URL: https://github.com/rayon-rs/rayon (visited on 08/01/2025).

[21] Doug Lea. “A Java fork/join framework”. In: *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*. Ed. by Dennis Gannon and Piyush Mehrotra. ACM, 2000, pp. 36–43. ISBN: 1-58113-288-3.

[22] K. C. Sivaramakrishnan et al. “Retrofitting parallelism onto OCaml”. In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 113:1–113:30.

[23] Domainslib Developers. *domainslib: Parallel Programming over Domains for Multicore OCaml*. 2026. URL: https://github.com/ocaml-multicore/domainslib (visited on 02/24/2026).

[24] Matteo Frigo et al. "Reducers and other Cilk++ hyperobjects". In: *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*. Ed. by Friedhelm Meyer auf der Heide and Michael A. Bender. ACM, 2009, pp. 79–90. ISBN: 978-1-60558-606-9.

[25] Umut A Acar, Arthur Charguéraud, and Mike Rainey. "Oracle-guided scheduling for parallel multithreaded programming". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 2011, pp. 161–172.

[26] Alexandros Tzannes et al. "Lazy scheduling: A runtime adaptive workaround for granularities". In: *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010.

[27] Shintaro Iwasaki and Kenjiro Taura. "A static cutoff for task parallel programs". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2016, pp. 139–150.

[28] Kyle Singer, Kunal Agrawal, and Tao B. Schardl. "Waste-Efficient Work Stealing". In: *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2026.

[29] Kunal Agrawal et al. "Adaptive scheduling with parallelism feedback". In: *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 100–109.

[30] Robert D Blumofe et al. "Cilk: An efficient multithreaded computing system". In: *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1995, pp. 207–216.

[31] Yuxiong He, Charles E Leiserson, and William M Leiserson. “The Cilkview scalability analyzer”. In: *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures.* 2010, pp. 145–156.