Work-Efficient Schedulers

Yue Yao

CMU-CS-19-130
Dec 2019

Computer Science Department
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Thesis Committee:
Umut A. Acar, *chair*
Randal E. Bryant

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Keywords: Work-Efficient Scheduler, Elastic Scheduler, $\alpha|\beta$-elastic Scheduler, Work-Effcient Work Stealing Scheduler, Elastic Work-Stealing Scheduler, Lifeline Forest, Concurrent Random Set, Parallel Scheduler

Abstract

The scheduling of multithreaded computations has attracted extensive research over past decades. Most of the research focused on design schedulers that are efficient in terms of runtime and space consumption, very often at the cost of performing more work than the computation itself required.

This work considers a new class of schedulers, called *work-efficient schedulers*. Work-efficient schedulers aim to minimize extra work, measured by the total number of instructions executed by all processors due to scheduling, including idle (referred to as spinning in this work) time. Specifically, the total amount of work performed during the scheduling of a computation must be within a small constant factor of the total work of the computation. This work first presents an offline *elastic* scheduler that achieves the goal by dynamically scaling up or down the processors it utilizes in response to the instantaneous parallelism. We prove a runtime and total work bound for our offline elastic scheduler and show that it achieves linear speedup with respect to the number of processors, while maintaining work efficiency.

This work further presents an online elastic work-stealing algorithm that aims at approximating the offline work-efficient schedulers. The elastic work-stealing scheduler augments the traditional work-stealing algorithm with a *lifeline forest* communication structure that allows processors to respond swiftly to varying instantaneous parallelism in a distributed manner. We implemente this algorithm then evaluate its performance and work efficiency by comparing it agaist existing implementations of traditional work-stealing schedulers. Results show that 1) for highly parallel computations, our elastic work-stealing scheduler is comparable to classic work stealing in its speedup; 2) for computations where parallelism is more limited, our elastic work-stealing scheduler performs considerably less work.

Acknowledgments

I want to thank my advisor Professor Umut Acar for his mentorship and support over the past year. This thesis would not have been possible without his guidance all along. His advice eventually grows into critical insights in my thesis. I would also like to thank Sam Westrick for providing insightful comments in every weekly meeting and for helping to bootstrap the project. I am also grateful for Professor Randy Bryant for being on my thesis committee and providing valuable feedback on my thesis. Finally, I would like to thank Professor Robert Harper for showing me the fantastic world of programming languages and his generous advice on life in academia and potential careers in research.

I received much support from family and friends in completing this research. My parents and grandparents have always been supportive not just for this work but also for me trying to accomplish the better self. Their support has always been a source of calm and confidence through the darkest hours in the past year. To my family, I want to extend my sincerest gratitude for your unwavering support. In the end, nothing calms me down more effectively in times of stress and uncertaintly than a family bond.

My friends Jiaqi Zuo and Yuning Zhang have also been a great source of academic inspiration as well as emotional support. Thank you for those fun discussions, whether it took place during lunch in iNoodle in GHC or during a dinner at Kaimei on Murray Avenue. You are both amazing friends.

Last but not least, I would like to thank Professor Dave Eckhardt for his continuous support and guidance through my Master's at CMU. Those 15-minute appointments not only addressed many of my concernes but also are genuinely fun. I also thank Tracy Farbacher for her kindness and patience over the past 18 months.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Shared memory multi-core or multiprocessor systems are pervasive in modern computing. Techniques for programming such systems have received extensive research, and *fine-grained* parallelism emerged as a promising approach. Fine-grained parallelism programming systems encourage the programmer to maximally identify independent computations, termed *threads*, and leave the details of executing those threads to the programming systems. This approach has proved to be very effective by Cilk [7], Intel's Threading Building Blocks [19], Java Fork/Join [17], X10 [22] and parallel ML [14].

One of the central challenges in designing and implementing fined grained parallel systems is to efficiently *schedule* threads on to processors. The scheduler dynamically map threads onto processors, hoping to achieve overall linear speedup with respect to the number of processors in the system. This is known as the (thread) scheduling problem. The algorithm that solves the scheduling problem is termed a (thread) *scheduler*. If the number of processors made available to the scheduler stays constant throughout the entire scheduling, then the scheduler targets a *dedicated environment*.

We may classify schedulers in terms of the information that they are allowed to take advantage of. *Offline schedulers* are omniscient in that they can see the entire system state, and they may schedule threads arbitrarily (e.g., move threads around, switch threads) without incurring any overhead. In particular, offline scheduler schedulers are aware of the structure of the entire computation from the beginning to the end, and they can utilize that information in any way they see fit. Studying offline schedulers allows us to draw conclusions about a particular scheduling algorithm given perfect information. *Online* *schedulers*, on the other hand, only have access to past and *local* information. In other words, the online scheduler can only make decisions based on the part of computation that it explored, or information obtained by explicit communications.

Previous research mainly focused on designing schedulers for better performance [1, 4, 9] and (or) efficient space utilization [6, 9]. Those schedulers try to provide as much speedup as possible often at the cost of *work-efficiency*: those schedulers often perform more work than explicitly demanded by the computation. The goal of this work is to design and implement a new class of schedulers, called ***work-efficient schedulers***. Informally, work efficient schedulers perform no more work than the total work of the computation during the entire scheduling.

Define the maximum number of processors a computation can utilize at some moment during scheduling as the *instantaneous parallelism* of that moment. Sources of work-***in***efficiencies include the overhead of scheduler code, communications between processors, and most importantly, processors *spinning* and trying to perform load balancing actions when there is limited instantaneous parallelism. As a concrete example, the work-stealing scheduler described in [4, 9] will arrange a processor to keep “stealing” even when there is no viable target, burning CPU cycles while making no progress on the computation itself. To achieve work-efficiency, we need to minimize the time processors spent on spinning when instantaneous parallelism is low, by allowing schedulers to dynamically varying the number of processors it utilizes and voluntarily give up processors if necessary. We call such schedulers *elastic schedulers*.

Elasticity is by no means a free lunch. Traditional schedulers arrange processors to keep performing load balancing actions when instantaneous parallelism is limited so that they can quickly pick up new threads when parallelism comes back. The ability to respond to varying instantaneous parallelism is termed *responsiveness*. Elasticity inevitably trades off responsiveness for work-efficiency, and a major challenge is to strike a balance between responsiveness and work-efficiency. This work presents a specific type of elastic offline scheduler. We show that this scheduler is both work-efficient and performant by proving a runtime and total work bound for computations scheduled using the scheduler.

This work further proposes an online scheduling algorithm that aims at approximating the offline work-efficient scheduler based on the traditional work-stealing scheduler. The randomized work-stealing algorithm is very successful both in theory [4, 9] and in practice [7, 10, 15]. The work-stealing scheduler is a distributed scheduling algorithm where each processor maintains and works on its own thread queue. Once a processor exhausted its queue, the processor obtains work by randomly picking a victim processor and try to migrate threads from the victim’s queue to its queue. The randomized, distributed nature of work-stealing makes it very robust and efficient. Our online scheduler, named ***elastic***

work-stealing scheduler, augments the traditional work-stealing algorithm by allowing for increasing or decreasing processors utilized during random steals. This allows us to scale the processor utilization in a randomized and distributed fashion.

The rest of this thesis is organized as follows:

- The rest of the Chapter 1 will first introduce to the readers the DAG model for modeling parallel computations. We will discuss a few related works. In section 1.4 we will present our main contributions.
- In Chapter 2 we will first augment the existing model for modeling offline schedulers to account for processors spinning. Then we will propose an elastic offline scheduler. With the help of the proposed scheduling model, we will prove that this algorithm is both work-efficient and performant by establishing both runtime bound and work bound for computations scheduled using this scheduler.
- In Chapter 3 we will describe the elastic work-stealing scheduler. The elastic work-stealing attempts to approximate the offline scheduler. We argue it is both elastic and work-efficient. We identify two key data structured utilized to implement our scheduler: ***lifeline forest*** and ***concurrent random set***. We describe algorithms that implement those data-structures and discussed their correctness and performance.
- In Chapter 4, we implement our elastic work-stealing scheduler and evaluate it by comparing it with existing implementations of traditional work-stealing algorithms. The results support that our scheduler is as performant as traditional work-stealing schedulers for highly parallel computations, and performances significantly less work for computations with limited parallelism. The data provide empirical evidence for the claims in Chapter 3.
- In Chapter 5 we conclude our work and briefly discuss future directions.

1.2 Multithreaded Computations as Computation Graphs

In this section we introduce a widely used [6, 8, 12] graph-theoretic model for analyzing multithreaded computations. The central idea is to model a multithreaded computation as an unfolding directed acyclic graph (***dag***) where nodes represent unit-time *instructions*, and edges represents dependencies between instructions.

Figure 1.1 demonstrates a computation graph of some parallel computation. We will introduce necessary terminologies and definitions using it as a concrete example.

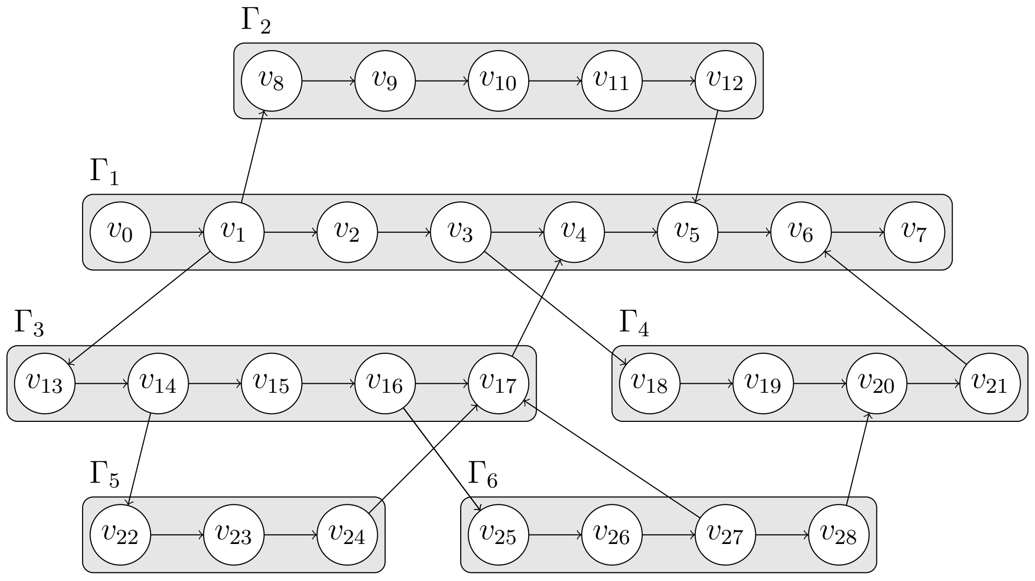


Figure 1.1: An exemplary computation dag G consisting 6 thread $\Gamma_1 \dots \Gamma_6$ and 28 nodes $v_1 \dots v_{28}$. v_0 is the *root* and Γ_1 is the initial thread. Computation starts with node v_0 in Γ_1 and terminates with v_7 in Γ_1. For this graph, $T_1 = 29$, and the span $T_\infty = 14$.

A computation must start with some instruction. In our case the computation begins with v_0. We call v_0 the ***root node***.

A ***thread*** is a sequence of connected nodes. In our example, the computation consists of six threads: each shaded region is a single thread. Because edges represents dependencies, nodes in a single thread form a sequential computation. The thread containing the root node is called the *initial thread*.

A ***P-processor schedule*** (or just *schedule*) of a computation graph is an assignment of nodes n to processors $p_1 \dots p_P$ for every time step i, and it must respect the dependencies specified by the graph. A schedule is valid if and only if for all nodes n, if n is scheduled at time t, then all predecessors of n have been scheduled before t. Figure 1.1 provided one possible 4-processor schedule of the computation.

At the beginning of time step t, the set of nodes whose dependencies are all satisfied is referred to as the set of *ready nodes* of time t. In other words, ready nodes are nodes that can be scheduled without violating dependencies at time t. Given two different threads Γ and Γ', if the execution of some node n of thread Γ at time step t puts node n' of thread Γ' in to the set of ready nodes at time $t + 1$, we say n enabled n'. In particular, if n' is the first node of Γ', we say n spawned n' (or equivalently Γ'). It is possible for a node to

Step	Ready nodes	Node assignment			
		p_1	p_2	p_3	p_4
1	$\{v_0\}$	v_0			
2	$\{v_1\}$		v_1		
3	$\{v_2, v_8, v_{13}\}$	v_2	v_8	v_{13}	
4	$\{v_3, v_9, v_{14}\}$	v_3	v_9	v_{14}	
5	$\{v_{18}, v_{10}, v_{15}, v_{22}\}$	v_{18}	v_{10}	v_{15}	v_{22}
6	$\{v_{19}, v_{11}, v_{16}, v_{23}\}$	v_{19}	v_{11}	v_{16}	v_{23}
7	$\{v_{12}, v_{25}, v_{24}\}$		v_{12}	v_{25}	v_{24}
8	$\{v_{26}\}$			v_{26}	
9	$\{v_{27}\}$			v_{27}	
10	$\{v_{28}, v_{17}\}$	v_{17}		v_{28}	
11	$\{v_{20}, v_4\}$	v_4		v_{20}	
12	$\{v_{21}, v_5\}$	v_5		v_{21}	
13	$\{v_6\}$	v_6			
14	$\{v_7\}$	v_7			

Table 1.1: A 4-processor schedule of the exemplary computation dag in Figure 1.1. The schedule takes exactly T_∞ steps to complete, reaching the theoretical limit. Adding more processor cannot improve runtime.

spawn multiple nodes. In our example, v_1 spawned v_8 and v_{13} at time step 2. On the other hand, it is also possible to have two nodes "simultaneously" enable the same node. In our example, execution of v_5 and v_{21} at time step 12 satisfied the dependencies of v_6. In this case, one can arbitrarily pick one of v_5 and v_{21} and claim it enabled v_6. Our arguments will still hold, regardless of the choice.

Another equivalent way of looking at this model is the following: we may think of a parallel schedule as iteratively removing executed zero in-degree nodes (ready nodes) from the computation graph, resulting in a smaller computation graph at each time step. In the beginning, only the root node has zero in-degree. At the end of the computation, all nodes have been removed, resulting in an empty computation graph.

the ***work*** of the computation, denoted by T_1, is defined to be the number of nodes in the graph. For this computation, $T_1 = 29$ since there are precisely 29 nodes. The work of computation is the number of steps a 1-processors schedule would take to complete the computation. In other work, it is the minimum amount of work one has to perform to complete the computation.

The ***span*** of the computation, denoted by T_∞, is defined to be the length of the critical path in the graph. For this computation, there exists one unique critical path, consisting $v_0, v_1, v_{13}, \ldots v_{16} \ldots v_{25} \ldots v_{28}, v_{20}, v_{21}, v_6, v_7$. That is $T_\infty = 14$ for this graph. The span of computation models the minimum time one would have to spend given infinite number of processors.

The ***average parallelism***, or just *parallelism* of the computation is defined to be quotient of work and span $\frac{T_1}{T_\infty}$. Conversely, we may define the number of ready nodes at time t as the ***instantaneous parallelism*** of time t. It measures the maximum number of processors the computation can effectively utilize at time t. For example, in our example, the instantaneous parallelism at time step 4 is 4, and it's only 1 at time step 8.

In our exemplary schedule in Table 1.1, we always try to schedule at many nodes as possible at each step. A scheduler that attempts to schedule as many ready nodes as possible at each time step is called a *greedy scheduler*. Note that our model does *not* mandate schedulers to be greedy to produce a valid schedule. Numerous previous work [11, 6] have stated and proved the following lemma using different models:

Lemma 1 (Runtime of greedy schedulers)**.** *Any P-processor schedule produced by a greedy scheduler of a computation graph with T_1 work and T_∞ span takes at most $T_1/P + T_\infty$ steps to complete.*

This model can be generalized [4] into the adaptive environment without much effort. In an adaptive setting, processors in this model are replaced with workers. The job schedulers select the "active" workers at each time step.

The model stated above is not sufficient for our purpose. In particular, the model says nothing about the processors that are not assigned with any node. Our idea is to augment the model to allow the scheduler to declare the amount of the processors it wishes to utilize at each time step. For each time step, the scheduler has to declare the number of processors it wishes to utilize, then assign nodes to a (potentially equal) subset of those processors. We refer to the number of processors a scheduler is willing to utilize at time step t as the *processor utilization* of time t.

1.3 Related Works

This work only considers the dedicated environment where the number of processors made available to the scheduler stays constant. Conversely, if the number of processors available varies, then the scheduler works in a *multiprogrammed environment* [4]. Scheduling in multiprogrammed environments is also known as *adaptive scheduling* [2]. Previous works [4, 3] proposed to model scheduling in an adaptive environment using a two-level scheduling model: there exists a *thread scheduler* (in some literature, *task schedulers*) that schedules threads on to a fixed set of *workers* (in some literature, *processes*), along with a potentially adversarial *job scheduler* that schedules a subset the workers onto the processors. In principle, the job scheduler corresponds to the operating systems' scheduler, and workers correspond to operating system threads. Researches focus on the design of thread schedulers.

In *Scheduling Multithreaded Computations by Work Stealing* by Blumofe et al., the authors first proposed the offline work-stealing algorithm for a dedicated environment. The authors prove that the algorithm is both space-efficient and performant. In particular, the author proved that the work-stealing scheduler takes on average $T_1/P + \mathcal{O}(T_\infty)$. Arora et al. in *Thread Scheduling for Multiprogrammed Multiprocessors* extended previous result to an online adaptive environment. Their work presented a non-blocking work-stealing scheduler implementation and proved that the work-stealing scheduler completes any computation of in $\mathcal{O}(T_1/P_A + (P/P_A)T_\infty)$ runtime, where P_A is the average processor availability provided by the job scheduler. Both works did provide an analysis of the total work performed, but one can show that the online scheduler could perform $\mathcal{O}(T_1 + PT_\infty)$ total work in a dedicated environment. In particular, for almost sequential computations where T_1/T_∞ is close to 1, the scheduler performs almost extra work proportional to the number of processors in the systems, even if we are not effectively utilizing them.

In *Adaptive Scheduling with Parallelism Feedback*, Agrawal et al. presented the A-

GREEDY offline scheduler for an adaptive environment. Their scheduler split the computations into *quantums* and explicitly communicate the number of processors it *desires* to the job scheduler at the beginning of each quantum. This is known as the *feedback* to the job scheduler. The desire is determined by measuring effective processor utilization, i.e., the average amount of processors spent on conducting computation, of the previous quantum. The A-GREEDY scheduler scales up and down the desire at an exponential rate, controlled by the responsiveness factor ρ. For an dedicated environment with quantums of unit size, the scheduler completes a computation in $T_1/P + 2T_\infty + \log_\rho P + 1$ steps. A similar bound was also derived for the adaptive environment. They also show that the scheduler can waste at most ρT_1 work. The goal of Agrawal's research was to prevent over-provisioning of the workers.

Our $\alpha|\beta$-elastic scheduler, being an offline scheduler for a dedicated environment, aims at work-efficiency. The notion of processor utilization in our scheduler is similar to the idea of parallelism feedback, namely "processor desire", in Argarwal's work. Both schedulers employed exponential scaling of processor utilization; however, our algorithm separately considers the up-scale and down-scale responsiveness. We show that they work in coordination to provide work-efficiency. Moreover, our algorithm is more conservative in that we do not increase processor utilization over the instantaneous parallelism. This conservative behavior is crucial for work-efficient scheduling.

In *Adaptive work stealing with parallelism feedback*, Agrawal et al. extended their previous work and presented an online work-stealing scheduler A-STEAL. The scheduler again splits the computation into quantums. In each quantum, the scheduler performs traditional work-stealing. Between quantums, the scheduler communicates its desire to the job scheduler and adjust the number of workers based on the processor allotment received from the job scheduler. Their work further provided proof on the runtime and work bound of the scheduler. Finally, their work presented and implementation of the scheduler and evaluated the implementation in a simulated environment.

Our elastic work-stealing scheduler is different from A-STEAL in several ways. Most importantly, our elastic scheduler is fully distributed. Every processor makes decisions based on its own local view of the system state. This allows our scheduler to avoid contention and synchronization at much as possible. On the other hand, we provided a implementation and evaluated it on a real system. Our work has a few limitations. We only considered the dedicated environment at this moment, and we did not provide formal proof of runtime or work efficiency. Overcoming those limitations would be the next step of our research.

Our work-stealing scheduler employs a communication structure between processors called *lifeline forest*. Informally, a lifeline is a channel between a sleeping processor and an active processor, through which the latter processor can send a message to wake up the former. The notion of a lifeline is first proposed by Saraswat et al. in *Lifeline-based global load balancing*. Their work proposed using a static, "low-degree, low-diameter, fully-connected" graph as a communication structure between nodes in a distributed system, to enable better work balancing and provide termination detection. Although the purpose of their work is different from ours, both works considered using work-stealing as a load balancing technique and allowing processors (nodes) to attach and "signal" lifelines during work-stealing as a method for conserving work.

1.4 Contributions

This thesis proposes two new schedulers: an offline $\alpha|\beta$-elastic work-efficient scheduler and an online elastic work-stealing scheduler, both for dedicated environments.

- We formalized a scheduling model for analyzing offline elastic schedulers for dedicated environments.
- Using the model we proposed, we show our $\alpha|\beta$-elastic scheduler is both work efficient and performant. In particular, we prove that in a P processor dedicated environment, for any computation of T_1 work and T_∞ span, the $\alpha|\beta$-elastic scheduler completes the computation in at most $\frac{T_1}{P}+T_\infty(1+\log_\alpha \beta)+\log_a P$ steps, performing at most $T_1\frac{\alpha\beta-1}{\alpha(\beta-1)}$ work.
- We designed an elastic work-stealing scheduler by augmenting the traditional work-stealing scheduler to allow for increasing (or decreasing) processor utilization during scheduling. Processors spontaneously maintain a dynamically varying *lifeline forest* to be responsive to varying instantaneous parallelism. We identify the *random concurrent set* as a critical data structure in implementing lifeline forests. We propose an algorithm for implementing random concurrent sets based on SNZI [13] trees.
- We implemented our elastic work-stealing scheduler and compared it against a traditional work-stealing scheduler. Evidence shows that our scheduler is both performant and work-efficient.

Chapter 2

Work-Efficient Offline Scheduler

2.1 A Model for Offline Elastic Scheduling

In this section, we will begin by introducing a model that enables us to account for varying processor utilization, namely elasticity, for offline schedulers. This model will be an extension to the well-recognized P-processor scheduling model introduced in section 1.2.

Suppose there are P processors numbered $1 \ldots P$ in the system. We will work model a parallel computation by considering what each processor will do for each time step. In each time step, the thread scheduler assigns each processor an *intention*, declaring whether this processor will be utilized, and how it will be utilized. Then the intentions of the processors are carried out, and we enter the next time step.

We formally define the model for offline elastic scheduling as follows:

- An P***-processor elastic schedule*** consists a sequence of time steps. For a time step i, denote the computation graph containing all unexecuted nodes as G_i. The schedule begins with $G_1 = G$ and terminates right at the end of time step t if G_t is empty. Round t is called the final *time step*.

- In each time step, the scheduler schedules a number of nodes onto P processors according to some *scheduling policy*. According the policy, it sets an *intention* $\mathcal{I}$ for each of the processor. An intention for a processor is one of three options:

  - A processor may decide to execute a node n, denoted as "$\mathsf{E}(n)$".
  - A processor may choose to sleep for this time step, denoted as "sleep".

  - A processor may be performing load balancing actions, denoted as "spin".

  The function from all processors to their intentions of time step i is called the *intention of time step i*, denoted using $\mathcal{I}_i$. A processor is said to be ***active*** if it's assigned an non-sleep intention, otherwise it's called ***inactive***.

  Different types of schedulers can be considered in this unified model by substituting different scheduling policies.

- All processors act out their intentions. As a result, if a node is executed by some processor as a result, then it's removed from G_i. Removing all executed nodes results in G_{i+1}. If a processor is active (executing, or spin), then one piece of work is performed.

The thread scheduler must respect data dependencies between nodes. Define the set of zero in-degree nodes $\{n_1, \ldots, n_r\}$ of G_i as the set of *ready nodes* at time step i, denoted as R_i. The scheduler may only execute nodes in R_i for time step i. In particular, $r_i \triangleq |R_i|$ is termed the *instantaneous parallelism of time step i*.

Step	Ready nodes	Utilization	Processor Intention			
			p_1	p_2	p_3	p_4
1	$\{v_0\}$	4	$\mathsf{E}(v_0)$	spin	spin	spin
2	$\{v_1\}$	2	spin	$\mathsf{E}(v_1)$	sleep	sleep
3	$\{v_2, v_8, v_{13}\}$	3	$\mathsf{E}(v_2)$	$\mathsf{E}(v_8)$	$\mathsf{E}(v_{13})$	sleep
4	$\{v_3, v_9, v_{14}\}$	3	$\mathsf{E}(v_3)$	$\mathsf{E}(v_9)$	$\mathsf{E}(v_{14})$	sleep
5	$\{v_{18}, v_{10}, v_{15}, v_{22}\}$	4	$\mathsf{E}(v_{18})$	$\mathsf{E}(v_{10})$	$\mathsf{E}(v_{15})$	$\mathsf{E}(v_{22})$
6	$\{v_{19}, v_{11}, v_{16}, v_{23}\}$	4	$\mathsf{E}(v_{19})$	$\mathsf{E}(v_{11})$	$\mathsf{E}(v_{16})$	$\mathsf{E}(v_{23})$
7	$\{v_{12}, v_{25}, v_{24}\}$	4	spin	$\mathsf{E}(v_{12})$	$\mathsf{E}(v_{25})$	$\mathsf{E}(v_{24})$
8	$\{v_{26}\}$	3	spin	sleep	$\mathsf{E}(v_{26})$	spin
9	$\{v_{27}\}$	2	spin	sleep	$\mathsf{E}(v_{27})$	sleep
10	$\{v_{28}, v_{17}\}$	2	$\mathsf{E}(v_{17})$	sleep	$\mathsf{E}(v_{28})$	sleep
11	$\{v_{20}, v_4\}$	3	$\mathsf{E}(v_4)$	spin	$\mathsf{E}(v_{20})$	sleep
12	$\{v_{21}, v_5\}$	3	$\mathsf{E}(v_5)$	spin	$\mathsf{E}(v_{21})$	sleep
13	$\{v_6\}$	2	$\mathsf{E}(v_6)$	sleep	spin	sleep
14	$\{v_7\}$	1	$\mathsf{E}(v_7)$	sleep	sleep	sleep

Table 2.1: A 4-processor elastic schedule of the exemplary computation dag in Figure 1.1. The total work performed in this schedule is 40 units.

Table 2.1 provides an 4-processor elastic schedule for the exemplary computation dag in Figure 1.1. Essentially, this model allows us to discuss work by differentiating between idle and sleeping processors and idle yet spinning processors for each time step.

Definition 1. *For time step i, the number of active processors is termed the processor **utilization** of time step i, denoted using u_i.*

The we may define ***the total work*** of a P-processor elastic schedule consisting t time steps as the sum of the work performed in each time step, namely $\sum_{i=1}^{t} a_i$.

An ***elastic scheduling policy*** is a scheduling policy that dynamically changes its processor utilization. The ability to adjust processor utilization gives the thread scheduler a way to bound the work-cost it incurs. This is very important for the thread scheduler to achieve work efficiency.

As a concrete example, let's consider a traditional non-elastic greedy scheduler. A non-elastic greedy scheduler employs the following scheduling policy $\mathcal{X}^{\mathsf{i}}_{\mathsf{greedy}}$:

Scheduler Policy 1 (Non-elastic Greedy Scheduler)**.** *For every time step i, suppose $R_i = \{r_1, \ldots, r_{r_i}\}$, assign nodes $r_1, \ldots, r_{\min(P, r_i)}$ to the first $\min(P, r_i)$ processors. The rest of the processors simply stays* **spin***.*

We once again prove the well known greedy scheduling lemma:

Lemma 2 (Greedy Scheduling Runtime Bound)**.** *For a computation graph of T_1 work and T_∞ span, the P-processor elastic schedule with the greedy scheduler completes in at most $T_1/P + T_\infty$ steps.*

Proof. We briefly go through the proof because the result is fairly well known. We classify time steps according to their instantaneous parallelism. For time step i, if $r_i \geq P$, then P nodes are removed from G_i. There can be at most T_1/P such time steps. Otherwise $r_i < P$. In this case, all zero in-degree nodes in G_i will be removed, reducing the length of the critical path by 1. Since the maximum length of the critical path is T_∞, there can be at most T_∞ such time steps. Combining both components, we see that there can be at most $T_1/P + T_\infty$ time steps. □

Since the scheduler is not elastic, we will incur exactly P units of cost under $\mathcal{C}_C$ each time step. Then the non-elastic greedy scheduler performs at most $T_1 + PT_\infty$ work, which is PT_∞ extra work.

We may also consider the "elastic version" of the traditional greedy scheduler.

Scheduler Policy 2 (Fully Elastic Greedy Scheduler)**.** *For every time step i, suppose $R_i = \{r_1, \dots, r_{r_i}\}$, assign nodes $r_1, \dots, r_{\min(P, r_i)}$ to the first $\min(P, r_i)$ processors. The rest of the processors stay* sleep *instead of spinning.*

For any computation graph, the runtime of its P-processor elastic schedule using the fully elastic greedy scheduler is identical to that with the non-elastic greedy scheduler. However, this scheduler is work-efficient because this scheduler performs exactly T_1 work, given that we never put processors on spin. The problem with it is this scheduler requires *full elasticity*, i.e., the scheduler will have to be able to respond to an arbitrarily dramatic change in instantaneous parallelism in just one time step.

Full elasticity is very hard to achieve in practice. In practice, schedulers schedule tasks onto worker threads, which usually are operating systems threads. Schedulers increase and decrease the processor utilization by increasing and decreasing the number of workers, either by blocking and unblocking them, or through thread creation functions such as `pthread_create`. All of those approaches costs CPU cycles to add or remove workers, limiting the elasticity one can possibly obtain. The following sections of this work propose and demonstrate a scheduler that is both work-efficient, almost equally performant, and most importantly, requires only limited elasticity.

2.2 $\alpha|\beta$-Elastic Greedy Scheduler

In this section, we will introduce the $\alpha|\beta$-elastic greedy scheduler. This scheduler is a greedy scheduler with limited elasticity characterized by two parameters α and β. We show that such simple constraint is enough to guarantee work efficiency while having little impact on scheduler performance in terms of runtime. We begin by formally defining $\alpha|\beta$-elasticity:

Definition 2 ($\alpha|\beta$-elasticity)**.** *Given two real-valued parameters $\alpha > 1$, $\beta > 1$, a scheduler is called $\alpha|\beta$-elastic if and only if, there exists a function $\hat{u}(i)$ such that $u_i = \lfloor \hat{u}(i) \rfloor$ and for all consecutive time steps i and $i+1$ where time step $i+1$ is not the final time step, then $\hat{u}(i)/\beta \le \hat{u}(i+1) \le \alpha\hat{u}(i)$. In other word, processor utilization never increase faster than α-fold or decrease faster than β-fold.*

The reason for this rather verbose definition of $\alpha|\beta$-elasticity is since α and β are real valued parameters, however processor utilization for each round is integer-valued. Now we present the $\alpha|\beta$-elastic greedy scheduler:

Scheduler Policy 3 ($\alpha|\beta$-elastic greedy scheduler)**.** *The $\alpha|\beta$-elastic greedy scheduler keeps track of a parameter $\tilde{u}_i \in \mathbb{R}$ for every time step s_i. Parameter $\tilde{u}_i$ serves as the upper bound for processor utilization for time step i. Let $\tilde{u}_0 \triangleq 1$. It determines the utilization $\tilde{u}_i$ for the current time step according to the following rules:*

Down-scale *If $r < \tilde{u}_{i-1}$, then set $\tilde{u}_i = \max(r, \tilde{u}_{i-1}/\beta)$.*

Satisfied *If $r = \tilde{u}_{i-1}$, then keep $\tilde{u}_i = \tilde{u}_{i-1}$.*

Saturated *If $r > \tilde{u}_{i-1} = P$, then keep $\tilde{u}_i = \tilde{u}_{i-1} = P$.*

Up-scale *If $r > \tilde{u}_{i-1}$ and $\tilde{u}_{i-1} < P$, then set $\tilde{u}_i = \min(r, P, \alpha\tilde{u}_{i-1})$.*

For the final time step, since $r = 0$, then the scheduler will simply carry out the down-scale rule.

Once the scheduler determined the $\tilde{u}_i$ for current time step, then it will utilize exactly $u_i = \lfloor \tilde{u}_i \rfloor$ processors and greedily schedule ready nodes on to those processors. The active processors without assigned nodes will simply spin*.*

Remark 1. *The following statements hold for the $\alpha|\beta$-elastic greedy scheduler:*

- *For any time step i other than the final time step, $1 \leq \tilde{u}_i \leq P$.*
- *The $\alpha|\beta$-elastic greedy scheduler defined matches up with our previous definition for $\alpha|\beta$-elasticity, with the function $\hat{u}(i)$ taken to be $\hat{u}(i) = \tilde{u}_i$.*

The $\alpha|\beta$-elastic greedy scheduler utilizes the elasticity to dynamically to changing instantaneous parallelism. If the instantaneous parallelism is less than processors available in the previous time step, the scale-up rule increases the desired number of processors at a maximum rate of α. If the instantaneous parallelism is more than processors available in the previous time step, the scale down rules decreases the desired number of processors at a maximum rate of β. In particular, it does *not* attempt to *predict* future instantaneous parallelism, meaning it will not increase or decrease processor utilization over the available instantaneous parallelism.

Next section, we show that the $\alpha|\beta$-elastic greedy scheduler is both performant and work-efficient in the sense that it achieves linear speed-up while performing very little extra work regardless of the number of processors in the system. This boils down to the following two theorems:

Theorem 1 (Runtime bound of The $\alpha|\beta$-elastic greedy scheduler)**.** *Any P-processor elastic scheduling of computation graph with T_1 work and T_∞ span using the $\alpha|\beta$-elastic greedy scheduler completes in at most $\frac{T_1}{P} + T_\infty(1 + \log_\alpha \beta) + \log_\alpha P$ time steps.*

Theorem 2 (Total work bound of The $\alpha|\beta$-elastic greedy scheduler)**.** *Any P-processor elastic scheduling of computation graph of T_1 work using the $\alpha|\beta$-elastic greedy scheduler completes with performing at most $T_1 \frac{\alpha\beta - 1}{\alpha(\beta - 1)}$ work.*

2.3 Analysis of the $\alpha|\beta$-Elastic Greedy Scheduler

For the analysis, we further elaborate on the rules of our scheduler into the following six rules.

- For scale-down rule, $r < \tilde{u}_{i-1}$. Let $\tilde{u}_i = \max(r, \tilde{u}_{i-1}/\beta)$. All r ready nodes are scheduled.

  A. If $r = \tilde{u}_i$. Parallelism drops within β rate. $\tilde{u}_i < \tilde{u}_{i-1}/\beta$. All ready nodes are scheduled.

  B. If $r < \tilde{u}_i$. Parallelism drops at a rate higher than β. $\lfloor \tilde{u}_i \rfloor - r$ processors are spinning. $\tilde{u}_i = \tilde{u}_{i-1}/\beta$. All ready nodes are scheduled.

C. $r = \tilde{u}_{i-1}$: Remains unchanged. All r ready nodes are scheduled.

D. $r > \tilde{u}_{i-1} = P$: Remains unchanged.

- For the scale-up rule, $r > \tilde{u}_{i-1}$ and $\tilde{u}_{i-1} < P$. Let $\tilde{u}_i = \min(r, P, \alpha\tilde{u}_{i-1})$

  E. $r = \tilde{u}_i$, Parallelism increases within α rate. All ready nodes are scheduled. $\tilde{u}_i > \alpha\tilde{u}_{i-1}$

  F. $r > \tilde{u}_i$, Parallelism increases at a rater higher than α. $\tilde{u}_i = \alpha\tilde{u}_{i-1}$. $r - \lfloor \tilde{u}_i \rfloor$ ready nodes are left for future time steps to execute.

Because $\tilde{u}_i \leq P$, rules A to F are mutual exclusive and covers all cases.

We would like to make the following remark, which is just a reiteration of the greedy scheduling lemma in our context:

Remark 2. *In a dedicated environment, if the time step i is governed by rule any one of the rules A, B, C or E, then span of the remaining graph decrease by 1, because all zero in-degree nodes are executed.*

2.3.1 Runtime Bound of $\alpha|\beta$-Elastic Greedy Scheduler

In this section, we provide a proof of Theorem 1. We will categorize each time step in the scheduling according to the scheduling rule applied to that time step.

The proof proceeds by categorizing the time steps according whether that time step is governed by rule D. We will bound the positive and negative cases separately and combine the counts in the end.

Lemma 3. *There are at most T_1/P time steps governed by D.*

Proof. Each time step governed by D decrease the size of G by P. Note that $|G| = T_1$ by definition, therefore there may be at most T_1/P time steps governed by D. $\square$

In the next paragraph will bound the number of occurrences of all other rules through a potential function argument. Define the potential function

$$\Phi(i) \triangleq T_\infty^i(1 + \log_\alpha \beta) - \log_\alpha \tilde{u}_{i-1}$$

for time step i, where T_∞^i is span of graph G_i. Consider the consecutive difference:

$$\Phi(i) - \Phi(i+1) = (T_\infty^i - T_\infty^{i+1})(1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} \tag{2.1}$$

Lemma 4. *The consecutive difference $\Phi(i) - \Phi(i+1)$ is non-negative for all rules, and decrease by at least 1 for all rules other than D.*

Proof. Proof by casing on all rules for time step i:

Rule A & B For rule A, $T_\infty^i - T_\infty^{i+1} = 1$, $\tilde{u}_i \geq \tilde{u}_{i-1}/\beta$. Then

$$\Phi(i) - \Phi(i+1) = (1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} \geq (1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_{i-1}/\beta} = 1$$

Rule C For rule C, $T_\infty^i - T_\infty^{i+1} = 1$, $\tilde{u}_i = \tilde{u}_{i-1}$

$$\Phi(i) - \Phi(i+1) = (T_\infty^i - T_\infty^{i+1})(1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} = 1 + \log_\alpha \beta > 1$$

Rule D For rule D, $\tilde{u}_i = \tilde{u}_{i-1}$. $T_\infty^i \geq T_\infty^{i+1}$. Then

$$\Phi(i) - \Phi(i+1) = (T_\infty^i - T_\infty^{i+1})(1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} = (T_\infty^i - T_\infty^{i+1})(1 + \log_\alpha \beta) \geq 0$$

Rule E For rule E, $T^i_\infty - T^{i+1}_\infty = 1$, $\tilde{u}_i > \tilde{u}_{i-1}$. Then

$$\Phi(i) - \Phi(i+1) = (T^i_\infty - T^{i+1}_\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} \geq (1 + \log_\alpha \beta) + \log_\alpha \frac{\tilde{u}_i}{\tilde{u}_{i-1}} \geq 1$$

Rule F For rule F, $T^i_\infty - T^{i+1}_\infty \geq 0$, $\tilde{u}_i = \alpha \tilde{u}_{i-1}$. Then

$$\Phi(i) - \Phi(i+1) = (T^i_\infty - T^{i+1}_\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{\tilde{u}_{i-1}}{\tilde{u}_i} \geq 0 - \log_\alpha \frac{\tilde{u}_{i-1}}{\alpha \tilde{u}_i} = 1$$

We have analyzed all cases. □

The analysis on consecutive difference allows us to bound the number of non-D-time steps by the potential function:

Lemma 5. *There are at most $T_\infty(1 + \log_\alpha \beta) + \log_\alpha P$ steps not governed by D.*

Proof. For the first time step, $T^1_\infty = T_\infty$ and $\tilde{u}_0 = 1$ by definition.

$$\Phi(1) = T^i_\infty(1 + \log_\alpha \beta) - \log_\alpha \tilde{u}_0 = T_\infty(1 + \log_\alpha \beta)$$

Suppose the computation finishes after t time steps. As noted in Remark 1, $\tilde{u}_{t-1} \leq P$. At the end of the computation, $T^t_\infty = 0$. Then

$$\Phi(t) = T^t_\infty(1 + \log_\alpha \beta) - \log_\alpha \tilde{u}_{t-1} \leq -\log_\alpha P$$

Then the potential difference is:

$$\Phi(1) - \Phi(t) = T_\infty(1 + \log_\alpha \beta) + \log_\alpha P$$

Since each time step not governed by D decreases the potential by at least 1, and the potential decreases monotonically for all time steps, therefore they may be at most
$T_\infty(1 + \log_\alpha \beta) + \log_\alpha P$ times steps not governed by rule D. □

The main result is now directly derivable:

Proof. Combining the results of lemma 3 and lemma 5 allows us to conclude theorem 1. A time step is either a governed by D or otherwise, therefore there are at most
$\dfrac{T_1}{P} + T_\infty(1 + \log_\alpha \beta) + \log_\alpha P$ time steps. This completes our proof for the performance bound. □

Theorem 1 suggests our scheduler may take at most $T_\infty \log_\alpha \beta + \log_\alpha P$ time steps compared to a greedy scheduler. We would like to provide some insight into the difference.

The bound contains the additive term $\log_\alpha P$. This term exists because we initialized the utilization $\tilde{u}_0 = 1$. Even if the computation is highly parallel, the scheduler still needs $\log_\alpha P$ time steps to ramp up.

The other term $T_\infty \log_\alpha \beta$ reflects our schedulers' behavior in the face of rapidly changing parallelism. In the extreme case where $\alpha \to \infty$, the term vanishes. This corresponds to the case where we may bring back processors as fast as we want. In the usual case, the size of this term is bounded by the down-scale factor β. This might be surprising at first glance. In fact, β controls how susceptible our algorithm is to an adversarial computation graph may be. A large β allows the processor utilization to drop quickly, which will take more time steps to ramp up again. A small β on the other hand, "smooth" out the rapidly changing instantaneous parallelism.

2.3.2 Total-Work Bound of the $\alpha|\beta$-Elastic Greedy Scheduler

In this subsection, we provide a proof of Theorem 2. Again we will employ an potential function argument. Consider the following potential function:

$$\Phi(i) = T_1^i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}\tilde{u}_{i-1}$$

where T_1^i is number of nodes in G_i. Consider the consider the consecutive difference $\Phi(i) - \Phi(i+1)$:

$$\Phi(i) - \Phi(i+1) = (T_1^i - T_1^{i+1})(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)$$

Lemma 6. *The consecutive difference* $\Phi(i) - \Phi(i+1) \geq \tilde{u}_i$ *for all time steps* i*. i.e.,* $\Phi(i)$ *decrease by at least* $\tilde{u}_i$ *for every time steps.*

Proof. Proof by casing on all rules of for time step i:

Rule A $T_1^i - T_1^{i+1} = \tilde{u}_i$ and $\tilde{u}_{i-1} > \tilde{u}_i$. Then

$$\begin{aligned}\Phi(i) - \Phi(i+1) &= (T_1^i - T_1^{i+1})(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &= \tilde{u}_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &> \tilde{u}_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + 0\\ &> \tilde{u}_i(1 + 0) = \tilde{u}_i\end{aligned}$$

Rule B $\tilde{u}_i = \tilde{u}_{i-1}/\beta$, which is $\tilde{u}_{i-1} = \beta\tilde{u}_i$. $T_1^i - T_1^{i+1} > 0$. Then

$$\begin{aligned}\Phi(i) - \Phi(i+1) &= (T_1^i - T_1^{i+1})(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &= \frac{1}{\beta - 1}(\beta\tilde{u}_i - \tilde{u}_i) = \tilde{u}_i\end{aligned}$$

Rule C & D In both cases, $T_1^i - T_1^{i+1} = r = \tilde{u}_{i-1}$, and $\tilde{u}_i = \tilde{u}_{i-1}$ Then

$$\Phi(i) - \Phi(i+1) = \tilde{u}_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) > \tilde{u}_i$$

Rule E & F In these cases $T_1^i - T_1^{i+1} = \tilde{u}_i$ and $\tilde{u}_{i-1} < \tilde{u}_i \leq \alpha\tilde{u}_{i-1}$. Then

$$\begin{aligned}\Phi(i) - \Phi(i+1) &= (T_1^i - T_1^{i+1})(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &= \tilde{u}_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &= \tilde{u}_i + \tilde{u}_i\frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1} + \frac{1}{\beta - 1}(\tilde{u}_{i-1} - \tilde{u}_i)\\ &= \tilde{u}_i + (\tilde{u}_i\frac{\alpha - 1}{\alpha} + \tilde{u}_{i-1} - \tilde{u}_i)\frac{1}{\beta - 1}\\ &= \tilde{u}_i + (\tilde{u}_{i-1} - \tilde{u}_i\frac{1}{\alpha})\frac{1}{\beta - 1}\end{aligned}$$

We have noted earlier, $\tilde{u}_i \leq \alpha\tilde{u}_{i-1}$. I.e., $\tilde{u}_i/\alpha \leq \tilde{u}_{i-1}$. Rewrite the result:

$$\Phi(i) - \Phi(i+1) \geq \tilde{u}_i + (0)\frac{1}{\beta - 1} = \tilde{u}_i$$

We have analyzed all cases. □

Now we are ready to prove Theorem 2.

Proof. Suppose the computation terminates at time step t, that is $W_t = 0$. By lemma 1, $\tilde{u}_{t-1} \geq 1$. Then the initial potential and terminal potential:

$$\Phi(1) = T_1^1(1 + \frac{\alpha - 1}{\alpha}\frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1}\tilde{u}_0 = T_1(1 + \frac{\alpha - 1}{\alpha}\frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1}$$
$$\Phi(t) = T_1^t(1 + \frac{\alpha - 1}{\alpha}\frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1}\tilde{u}_{t-1} = 0 + \frac{\beta}{\beta - 1}\tilde{u}_{t-1} \geq \frac{\beta}{\beta - 1}$$

Then potential difference

$$\Phi(1) - \Phi(t) \leq T_1(1 + \frac{\alpha - 1}{\alpha}\frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1} - \frac{\beta}{\beta - 1} = T_1(1 + \frac{\alpha - 1}{\alpha}\frac{\beta}{\beta - 1})$$

On the other hand, by lemma 6,

$$\Phi(i) - \Phi(t) = \sum_{j=1}^{t-1}(\Phi(j) - \Phi(j+1)) \geq \sum_{i=1}^{t-1}\tilde{u}_j$$

As Remark 1 has noted, $u_i = \lfloor \tilde{u}_i \rfloor$. Then the total cost incurred under cost rule $\mathcal{C}_C$ is

$$\sum_{i=1}^{t-1} a_i \leq \sum_{i=1}^{t-1}\tilde{u}_i \leq \Phi(i) - \Phi(t) = T_1(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) = T_1\frac{\alpha\beta - 1}{\alpha(\beta - 1)}$$

We have concluded our proof of Theorem 2. □

According to the theorem, our scheduler performs at most $T_\infty\frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}$ work. We would like to make the following remarks regarding this bound.

First of all, the bound makes sense for extreme values of α and β. If $\beta \to \infty$, the extra work approaches zero. If $\alpha \to \infty$, we perform at most $T_\infty\frac{1}{\beta - 1}$ extra work.

Moreover, the extra work is bounded by $\frac{T_\infty}{\beta - 1}$ regardless of the choice of α. For sufficiently large β, the constant is very small. The bound shows that our algorithm is work-efficient.

Finally, decreasing α also helps to improve work-efficiency. For example, if $\alpha = \beta = 2$, we ends up perform at most half of T_1.

Chapter 3

Elastic Working-Stealing Scheduler

3.1 Introduction

In this chapter, we will present an elastic algorithm that aims at approximating the offline $\alpha|\beta$-elastic greedy scheduler. The scheduler allows the processors to respond to the changing parallelism by disabling themselves and enabling each other through a dynamically varying communication structure between the processors. We further argue that such design achieves exponential rate up-scaling and down-scaling. We will identify critical data structures to maintain the communication structure and provide support for random stealing in the face of varying enabled processors. We will further discuss possible implementations of these data structures.

We will kick off our discussion by briefly introducing the traditional work-stealing scheduler. Our scheduler will augment the traditional work-stealing algorithm by changing the stealing part of the algorithm. Here first briefly reiterate the work-stealing scheduler, as described in the work [9] of Blumofe et. al.

For every processor p in the system, the algorithm maintains a doubly-ended work queue Q_p. Processors push and pop tasks from/to their own work queue at the bottom. They remove tasks from others' work queue from the top.

Spawns If the task under execution spawns another task, then the processor begins to work on the child task, pushing the parent task into its work queue from the bottom.

Dies or Stalls If the current task dies or stalls, the processor first attempts to pop a task from the bottom of its work queue. If the work queue is empty. The processor will try to obtain tasks from other processors through *work stealing*. Details of work-stealing will be discussed later on.

Enables If the task happens to enable another task, the enabled task is pushed to the bottom of the processor's own work queue.

In the beginning, the root task is assigned to an arbitrary processor. All other processors start with work-stealing.

When a processor engages in work-stealing, the processor becomes a *thief*. The thief uniformly randomly chooses another processor in the system. The chosen processor is termed the *victim*. The thief then attempts to pop the work queue of the victim from the top. If the victim's work queue is non-empty, then the operation succeeds, and the thief starts working on the task it just obtained. This terminates the work-stealing phase. If the victim's work queue is non-empty, then the thief simply retries by randomly choosing another victim.

3.2 Elastic Work-Stealing with Lifeline Forest

The goal of our algorithm is to augment the work-stealing algorithm so that processors may disable themselves when the instantaneous parallelism is low and wake each other up when parallelism comes back. To reduce contention, decisions to wake up or put processors to sleep should be made in a distributed fashion. The elastic work-stealing algorithm achieves this purpose with two simple heuristics:

- When a processor made a few failed steal attempts, the processor can be confident that the instantaneous parallelism is low, and it should go to sleep.
- When a stealing processor encounters another stealing processor, it's more work-efficient to invite the other processor to steal "on its behalf". In particular, the former processor could ask disable itself and ask the latter processor to wake it up when parallelism comes back.

Those two simple heuristics become the foundation of our elastic work-stealing scheduler. The elastic work-stealing scheduler augments the work-stealing algorithm by allowing the processors to sleep and (or) wake up other processors during work-stealing. In the elastic work-stealing scheduler, when a processor becomes a *thief*, it uniformly randomly chooses another *currently active* processor. The chosen processor is termed the *victim*. The thief then attempts to pop the work queue of the victim from the top as usual.

- If the victim's work queue is empty and the victim is also stealing. Then the thief will try to set up a message channel between itself and the victim and ask the victim to message him in the future (explained shortly after). If the operations succeed, then it sleeps by waiting for a message on the channel. We say the thief now ***depends on*** the victim.

- If the victim's work queue is non-empty, then the thief will obtain work. The thief wakes all processors depending on it (processes who previously requested its "help") by sending a message through each of the previously established channels, removing those channels at the same time.

For dependent processors, sending a message through the channel they previously attached is the only way to wake them up. The channel is figuratively called a ***lifeline***. The terminology is inspired by the work [20] of Saraswat et. al. In Saraswat's work, they considered a fixed grid-shaped lifeline structure for distributed systems. Our algorithm, on the other hand, features a dynamically forming and destructing communication structure.

For our algorithm to work, we must ensure there are no loops in the lifeline structure. In other words, if we treat each processor as a node, the nodes and lifelines form a *forest*. In the next paragraph, we will introduce a new data structure called ***lifeline forest***, which manages the communication structure between processors.

A lifeline forest is a forest (a number of trees) with a fixed number of nodes (called *endpoints*). An edge from one endpoint to another corresponds to a lifeline from former to the latter (the latter is responsible for sending the message). Roots of the forest are the zero out-degree endpoints. Those endpoints are called *independent* endpoints. When a lifeline is formed, one of the independent nodes becomes a child of another, merging two trees into one. Conversely, when a lifeline is removed, one tree splits into two trees.

We may define the operations for lifeline forest as follows:

Data Structure 1. *A lifeline forest is a concurrent data structure with the following operations:*

`new(n)`	*Creates an lifeline forest with n endpoints. Endpoints are conveniently number from $1 \ldots n$. An empty lifeline forest contains no edges.*
`L.attach(p, v)`	*Attempt to attach a lifeline from endpoint p to endpoint v. If the operation succeeds if and only if both p and v are independent and different. Returns whether the operation is successful.*

`L.signal(v)`	*If endpoint v is dependent, then nothing happens. Otherwise, endpoint v sends a message to all lifelines attached to it, removing those lifelines in the process.*
`L.wait(p)`	*If the endpoint p is independent, then nothing happens. Otherwise, the process "blocks" (that is, sleeps and waits) for a message from the lifeline p previously attached.*
`L.sample()`	***Uniformly** randomly return an independent endpoint. If such p does not exists, return None to signify failure.*

Operations `attach`, `wait` and `signal` are linearizable.

We can implement our elastic work-stealing algorithm with the help of a lifeline tree in a straight forward fashion. Algorithm 1 presents the entire elastic work-stealing algorithm in terms of the lifeline forest data structure. The key idea is that we assign each processor $1 \ldots P$ an endpoint, conveniently also numbered $1 \ldots P$. Then each endpoint in the lifeline forest becomes the corresponding processor's "mailbox".

It is possible for the `sample` operation on the lifeline forest to return the argument p. Clearly, trying to steal from oneself does not make progress for the scheduling algorithm. However, after very few retries, the `sample` operations will always eventually return a victim other p because there always exists at least one processor that is working on some task. In other words, there is at least one processor, other than the thief, that is independent. It will only take the thief a few retries to find that processor (or another victim).

The algorithm presented here assumes that the computation graph has a maximum out-degree of two. This corresponds to the reality that many existing programming systems [7] provide only binary forking primitives. However, the validity of our algorithm does not depend on binary forking. Extending the algorithm from binary forking to multi-ary forking can be achieved by changing line 6 so that the processor picks an arbitrary task from the set of new tasks, pushes n_p along with the rest of new tasks on to Q_p. In the next line, it set n_p as the chosen task.

We argue that the elastic work-stealing algorithm approximates the online algorithm by arguing it exhibits exponential scaling of processor utilization in the face of varying parallelism. When separately consider rapidly decreasing instantaneous parallelism and rapidly increasing parallelism.

- If the parallelism is quickly decreasing, because the probability for a thief to find another stealing processor is proportional to the number of thieves in the systems, a

Algorithm 1: Elastic Work-Stealing Scheduler

```
Data: A lifeline forest $L$ initialized with the set of all $P$ processors.
Data: For each processor $p$, an empty queue $Q_p$ and its assigned task $n_p$
/* Set off the computation                                    */
Set $n_0$ to be the root task, all other $n_p$ to None;
/* Scheduling loop for each processor $p$.                    */
while True do
    while $n_p \neq$ None do
        $newTask \leftarrow$ Execute($n_p$);
        if $newTask \neq$ None then
            pushBotom($Q_p$, $n_p$);
            $n_p \leftarrow newTask$;
            continue;
        end
        $n_p \leftarrow$ popBottom($Q_p$);
    end
    while True do
        $shouldSleep \leftarrow$ False;
        $victim \leftarrow L$.sample();
        $n_p \leftarrow$ popTop($Q_{victim}$);
        if $n_p \neq$ None then
            $L$.signal($p$);
        else
            atomic
                if $n_{victim} =$ None then
                    // The victim is stealing or sleeping.
                    $shouldSleep \leftarrow L$.attach($p$, $victim$);
                end
            end
        end
        if $shouldSleep$ then
            $L$.wait($p$) // Put myself to sleep.
        end
    end
end
```

thief is very likely to find another thief. As a result of the steal, either the thief or the victim is disabled. In other words, the rate at which the number of idle processors decreases is proportional to the number of idle processors, which suggests an exponential decrease. In other words, when the parallelism is low, the lifeline forest "folds" at an exponential rate.

- When the parallelism comes back, the algorithm starts to "unfold" the lifeline forest. Suppose there is plenty of parallelism in the system, then almost all steals succeed and further unfolds the lifeline forest. Notice that the unfolding of the lifeline tree simply the "reverse" of the formation of lifeline forest. Thus we should observe an exponential increase in processor utilization.

The work efficient work-stealing algorithm requires careful treatment of synchronization. Because all Q_p are concurrent queues, their operations are assumed to be atomic. Since we are in the dedicated environment, operations do not have to be non-blocking. Blocks marked with `atomic` means the code enclosed must be executed in an atomic fashion. As we will see shortly after, atomicity is necessary to ensure the correctness of the algorithm.

The efficient work-stealing algorithm will is **safe** if it satisfies the following properties:

1. In all circumstances, there must not exist a lifeline that is attached to an executing processor. I.e., a processor executing lines 4 to line 10 must have no lifeline attached to it.

2. In no circumstances can a processor p_j becomes a parent of itself in the lifeline forest. In other words, nodes in the lifeline forest cannot form a loop.

Violating those properties cause us to lose available processors. Suppose p attached a lifeline to $victim$ that results in a violation property 1. Now p becomes an orphaned sleeping processor that potentially will never wake up (because $victim$ may always have tasks to do). We effectively lose one processor in this case. Violating property 2 will, in the worst case, causes us to lose all processors participating in the loop (along with all of their children).

Remark 3. *The work efficient work stealing algorithm described in algorithm 1 is safe.*

We argue that our algorithm is safe by examining each property separately.

Property 1 Consider the moment when p attach a lifeline to $victim$ on line 21 when it observes n_{victim} = `None` and the victim's Q_{victim} is empty. If the victim is in the scheduling loop, then it must between line 10 and the loop check. In either case, it will exit the loop, and we are fine. Otherwise, the victim is current stealing. If it is before line 15, then it will either continue stealing or remove the lifeline we just attached. Either way we are fine. If it's after line 15, then we know it will continue stealing because we know n_{victim} is `None`. We have examined all cases.

Property 2 Property 2 is automatically satisfied by the guarantees provided by the `attach` operation of the lifeline tree.

3.3 Lifeline Forest and Concurrent Random Set

It remains to discuss how to implement a lifeline forest described in Data Structure 1. One important responsibility of a lifeline forest is to keep track of the independent endpoints. We formulate the following data structure to achieve this purpose:

Data Structure 2. *A concurrent random set is a concurrent set where elements in the set is draw from a fixed finitely large set $\mathcal{I}$. In particular, suppose further elements in the set is draw from an integer index-able set $\mathcal{S}$. It supports the following operations:*

`new(n)`	*Creates an empty random set `rs` on a set of n possible elements.*
`rs.add(i)`	*Add the item i into the random set `rs`.*
`rs.exists(i)`	*Returns `True` if the item is in the set, `False` otherwise.*
`rs.remove(i)`	*Remove the item i from the random set `rs`.*
`rs.sample()`	***Uniformly** randomly return an element i in the set `rs`. If the set is empty, return None to signify failure.*

Ideally, the concurrent random set supports concurrent `add`, `remove` and `sample` without or with very low contention. How concurrent random set may be implemented is elaborated in later sections.

3.3.1 Lifeline Forest

A lifeline forest (without versioning) for a set of processors P consist of the following data fields:

`rs`	A concurrent random set `rs` initialized to contain all processors.
`endpoints[`n`]`	Data structure maintained for every endpoint n, described below.

And for each endpoint maintains the following data fields:

`sem`	A semaphore for every processor p, initialized to zero.
`lifelines`	A list of processors that attached to processor p.
`waitCnt`	An integer initialized to zero that counts how many processors are sleeping on the lifeline p attached earlier.

Semaphores allow the calling processors to block on a lifeline. The semaphore acts as a channel between the parent and its children in the lifeline forest: `sem_down` corresponds to waiting on the channel, and `sem_up` corresponds to sending a wake-up message through the channel. It's important to make sure that the wake-up message sent by the parent does not get lost in case the intended receiving process is not ready to wait on the endpoint yet. The semantics of Semaphores is ideal for our purpose.

Algorithm 2 formally defines the operations on a lifeline forest. Code sections marked as `atomic` must be implemented in a way that all effects are observed at once, assuming operations of the concurrent random set are atomic. Atomicity guarantees can be implemented trivially using locks in most systems. This algorithm also makes use of the standard compare-and-swap primitive (`CAS` in the pseudocode).

Now we define and prove the correctness for our algorithm. In our algorithm, an endpoint p is considered to depend on v if and only if p is an element of the list L.`lifelines[`v`]`. We now show that our algorithm satisfies our behavior specification *in a sequential setting*, first without worrying about the linearization requirement. Observe that our data structure maintains the following set of invariants:

1. An endpoint p is in the concurrent random set L.`rs` if and only if it's an independent endpoint. In other words, the concurrent random set L.`rs` always contains all independent endpoints.

Algorithm 2: Operations on Lifeline Forest without Versioning

```
Data: A lifeline forest $L$ initialized with the set of all processors.
fun attach($L$, $p$, $v$):
    atomic
        if $p \neq v$ and $L$.rs.exsists( $p$) and $L$.rs.exsists( $v$) then
            append($L$.nodes[$v$].lifelines, $p$);
            $L$.rs.remove ($p$);
            return True;
        end
        return False;
    end
end
fun signal($L$, $p$):
    atomic
        if not $L$.rs.exsists($p$) then return;
        foreach $v$ in $L$.nodes[$p$].lifelines do
            $L$.rs.add ($v$);
            for $i = 1$ to $L$.nodes[$v$].waitCnt do
                sem_up($L$.nodes[$v$].sem)
            end
            $L$.nodes[$v$].waitCnt $\leftarrow$ 0;
        end
        clear($L$.nodes[$p$].lifelines)
    end
end
fun sample($L$):
    $L$.rs.sample();
end
fun wait($L$, $p$):
    atomic
        $toWait$ $\leftarrow$ False;
        if not $L$.rs.exsists($p$) then
            $toWait$ $\leftarrow$ True;
            $L$.nodes[$p$].waitCnt += 1;
        end
    end
    if $toWait$ then sem_down($L$.nodes[$p$].sem);
end
```

2. An endpoint may attach at most one lifeline.
3. `waitCnt` of endpoint n is the number of processors blocked on n.
4. No processor can be blocked on an independent endpoint.
5. For all endpoints, `sem` is always zero-valued.

We can verify for a newly created lifeline forest that the invariants are satisfied. Invariant 2 allows us to refer to a specific lifeline by referring to the endpoint that attached it in the first place. We then continue our proof by separately analyze each operation:

`attach` If $p = v$ then the function returns `False`. If any one of p and v is dependent, then by invariant, it is not in the set `rs`. Then the function returns `False`. If both p and v are independent, then p is added to v's list of lifelines, becoming a dependent of v. The function finally returns `True`. Now we have shown `attach` satisfies the behavior specification. We then show that all invariants are still intact. We removed p from the set because p became dependent at the end. p may attach at most one lifeline because if p would be dependent at the beginning if p previously attached a lifeline. Since p is previously an independent endpoint, then no processor may be blocked at p. In other words `waitCnt` $= 0$ for p at the end. We have verified all invariants.

`signal` If p is a dependent endpoint, then by the invariant it's not in the set `rs`. In this case, nothing happens. Otherwise, p is an independent endpoint. By definition of independence, its `lifelines` contains all endpoints that depend on p. Take arbitrary dependent endpoint v. Its lifeline will be removed in the end if because `lifelines` of endpoint p is cleared. Since v can attach at most one lifeline, it becomes an independent endpoint when its only lifeline is removed. Because `waitCnt` counts the number of processors blocking on v through its semaphore, upping the semaphore `waitCnt` times wakes up all those processors, resulting in a zero-valued semaphore. Here we have established `signal` behaves as specified. It is left to argue that the invariants are preserved. All previously dependent endpoints, that according to the invariants have been removed from `rs` earlier, are added back as a result of `rs.add`. No processors will be blocked on any v, and the `waitCnt` is cleared. We have verified all invariants.

`wait` If p is independent then p is in the set. The function does nothing. Otherwise, p is dependent, and the calling processor will eventually be blocked at p because the semaphore is zero by invariant. By invariant `waitCnt` holds the number of processors currently blocked on p. Because calling processor will end up blocked at p, we have to increase `waitCnt` by 1. We have verified all invariants and `wait`'s behavior.

`sample` By Invariant 1, L.`rs` contains all independent processors, therefore `rs.sample` uniformly randomly select an independent processor by the specification of `rs.sample`.

It is left to argue that operations in Algorithm 2 is linearizable. Unfortunately, this is not the case, and it almost is. Here we present a case where Algorithm 2 fails to block a processor when there does exist a lifeline. Consider the following execution trace in a system with 2 processors p_1 and p_2.

- Initialize a lifeline forest with two endpoints n_1 and n_2.
- Processor p_1 attaches n_1 to n_2. The operations successes.
- Processor p_1 try to wait on n_1. Suppose p_1 is suspended right at line 34. At this moment, L.`waitcnt[`n_1`]` $= 1$.
- Suppose processor p_2 signals n_1. It will dismantle existing lifelines and increase the semaphore of n_1 to 1.
- Processor p_2 then reestablish lifeline by attaching n_1 to n_2.
- Finally processor p_2 waits on n_1. At this moment there exists a lifeline from n_1 to n_2, yet p_2 will not block on n_1 because the decrement on n_1's semaphore succeeds, violating the specification for `wait`.

This situation can occur because the newly established lifeline see the processors blocked at the "old" lifeline. In Algorithm 2, a newly established lifeline erroneously "inherits" the same L.`waitcnt[`n_1`]` from the previous lifeline. Algorithm 3 solves this problem by versioning the semaphore and `waitcnt`. Whenever a new lifeline is created, it increases the version number so that all other operations always operate on the latest lifeline, if there exists one. To implement a lifeline forest with versioning requires us to maintain the following data fields for each endpoint:

`v`	Version number of its own lifeline. Initialized to zero.
`sem[v]`	Semaphore for version v, initialized to zero.

Algorithm 3: Operations on Lifeline Forest with Versioning

```
Data: A lifeline forest $L$ initialized with the set of all processors.
fun attach($L$, $p$, $v$):
    atomic
        if $p \neq v$ and $L$.rs.exists($p$) and $L$.rs.exists($v$) then
            $ver \leftarrow L$.nodes[$p$].v + 1;
            sem_init($L$.nodes[$p$].sem[$ver$], 0);
            $L$.nodes[$v$].waitCnt[$ver$] $\leftarrow 0$;
            $L$.nodes[$v$].v $\leftarrow ver$;
            append($L$.nodes[$v$].lifelines, $p$);
            $L$.rs.remove($p$);
            return True;
        end
        return False;
    end
end
fun signal($L$, $p$):
    atomic
        if not $L$.rs.exists($p$) then return;
        $ver \leftarrow L$.nodes[$p$].v;
        foreach $v$ in $L$.nodes[$p$].lifelines do
            $L$.rs.add($v$);
            for $i = 1$ to $L$.nodes[$v$].waitCnt[$ver$] do
                sem_up($L$.nodes[$v$].sem[$ver$])
            end
        end
        clear($L$.nodes[$p$].lifelines)
    end
end
fun wait($L$, $p$):
    atomic
        $ver \leftarrow L$.nodes[$p$].v;
        $toWait \leftarrow$ False;
        if not $L$.rs.exists($p$) then
            $toWait \leftarrow$ True;
            $L$.nodes[$p$].waitCnt[$ver$] += 1;
        end
    end
    if $toWait$ then sem_down($L$.nodes[$p$].sem[$ver$]);
end
```

lifelines	A list of endpoint that attached a lifeline to endpoint p.
waitCnt[v]	Number of processors blocking on the lifeline of version v.

There is no change for `sample` operation and it's omitted for brevity.

We argue without a formal proof that Algorithm 3 satisfies our behavior specification. The algorithm is identical Algorithm 2, except that operations `signal` and `wait` now both accesses the latest version of `sem` and `waitCnt`. Previous proof of correctness still holds if we refer to `sem` and `waitCnt` by their latest version. Furthermore, the data structure is now linearizable. Linearization points of operations `attach` and `signal` are the exit points of their atomic regions. For `wait`, the linearization point is right before the `sem_down` procedure call.

3.3.2 SNZI Concurrent Random Set

In this section we will discuss how to design SNZI concurrent random set. We will first introduce two trivial solutions and discuss their properties. We will argue both of them are insufficient for our purpose. Then we will discuss a third *Scalable Non-Zero Indicator* (SNZI) based implementation, which is based on the work [13] on SNZI trees.

To remind remind the readers of what a concurrent random set is, here we reiterate the specifications on the operations it should support support. A concurrent random set is a concurrent data structure supporting the following operations:

new(n)	Create an empty random set `rs` on a set of n possible elements.
rs.add(i)	Add the item i into the random set `rs`.
rs.exists(i)	Returns `True` if the item is in the set, `False` otherwise.
rs.remove(i)	Remove the item i from the random set `rs`.
rs.sample()	**Uniformly** randomly return an element i in the set `rs`. If the set is empty, return None to signify failure.

As stated before, elements in a concurrent random set must be drawn from a fixed, predetermined, integer indexable set $\mathcal{S}$. The range of possible elements is termed the *space* of the elements, and the size of the space is denoted using S. Ideally, implementations should provide fast and contention-free access to the data structure and maintain linearizability for all operations. It turns out satisfying all those requirements at once is hard, and we are unable to identify such a solution. Instead, we study what requirements may be relaxed to allow for practical purposes. To be formal:

Definition 3 (Relaxed specification for `sample` operation)**.** *An implementation of a concurrent random set must at least guarantee the following properties for `sample` operation.*

- *If an element `i` is added to the set before some invocation of `sample` operation, and it is not the target any concurrently executing `remove` operation, it must be a possible return value of `sample` operation.*
- *If an element `i` is removed from the set before some invocation of `sample` operation, and it is not the target of any concurrently executing `add` operation, it can not be a possible return value of `sample` operation.*

The random set is utilized in two ways:

- When the parallelism is plenty, the random set is mainly used to select a victim uniformly randomly. Because there is high instantaneous parallelism, processors are very likely to find work through random steal, and write access to the random set is fairly infrequent. A good design should provide a contention-free `sample` in this case.
- When instantaneous parallelism starts to vary, processors frequently sleep or wake up depending on whether the parallelism is decreasing or increasing, it's important for `sample` provide an up-to-date response. We want to ensure processors do not waste effort trying to steal from an already sleeping processor during down scaling or trying to steal from a sleeping processor during up-scaling.

As we have noted before, the correctness of our algorithm does not rely on the `sample` always returning a consistent value. Allowing `sample` to return stale value rarely will in exchange for improved performance may be acceptable.

In the next few paragraphs, we provide three implementations that have experimented with. The first implementation is a bidirectional map from index to processor realized using compacted array. The second in implementation is just a simple array. The third implementation is based on SNZI (pronounced as *snazzy*) objects [13] proposed by Ellen et al.

Array Based Bidirectional Map (Array-BiMap) The first implementation maintains the following data strcture for the set:

`count`	Current number of elements in the set.
`elements[n]`	An array of size S. It's first `count` elements contains all elements in the set.
`locations[n]`	An array of size S, mapping values into their location in the set. If the element is not in the set, it's location is set to nullary value, which is -1 in our implementation.

The operations are straight forward.

`new(S)`	Creates an empty random set with `count` set to zero.
`rs.add(i)`	Adding an element i is done by appending i to `elements` array and setting corresponding entry in `locations` array, then increase `count`.
`rs.query(i)`	Returns if `locations[i]` is a nullary value.
`rs.remove(i)`	The idea is to swap the element removed with the last element in `elements` array. The `remove` operations ensures the `elements` array is always compact so that sampling from it is fast. See Algorithm 4 for details.
`rs.sample()`	Generate a random value p such that $0 \leq p <$ `count`, return `elements[p]`.

For this simple design, all operations complete in constant time. However operations `add`, `query` and `remove` must be synchronized to ensure correctness. Fortunately there is no need to synchronize `sample` to maintain linearity. This data structure is fast and contention-free when there is ample instantaneous parallelism in the system. However, it becomes a single point of contention when there exists when the algorithm is trying to scale the processor utilization, which is not an ideal implementation for our purpose.

In parctice, the complexity of `remove` operation made it hard to implement synchronization without using a lock. This makes it hard to devise non-blocking versions of this data-structure.

Algorithm 4: `remove` Operation for Array-BiMap Implementation

```
fun rsRemove(rs, i):
    atomic
        $rs$.count ← $rs$.count − 1;
        $l$ ← $rs$.locations[$i$];
        $v$ ← $rs$.elements[$rs$.count];
        $rs$.locations[$i$] ← −1;
        $rs$.elements[$l$] ← $v$;
        $rs$.locations[$v$] ← $l$;
    end
end
```

This design has one extra advantage. It is possible to ask the data structure to report the number of elements in the set at the beginning of `add` and `remove` operations using the return values of those operations. It can be easily achieved by simply returning `count` field at the end of both operations. This is not directly useful for our purpose, but it can be useful for some other application, as we will see later on.

Naive Array Implementation The second implementation uses just one array as the data structure:

`exists[n]`	An array of size S mapping values into a Boolean indicator, indicating whether the value is in the set.

Operations are implemented exactly as one would expect:

`new(S)`	Creates an empty random set with all entries of `exits` set to `False`.
`rs.add(i)`	Adding an element i is done by setting `exists[n]` to `True`.
`rs.query(i)`	Returns if `exists[i]` is `True`.
`rs.remove(i)`	Removing an element i is done by setting `exists[n]` to `False`.
`rs.sample()`	Pick a random element i from `exists` array. Then if `exists[e]` $=$ `True` then return e, otherwise we just try again.

This very simple algorithm has a number of nice properties. First of all, the data structure is contention-free in that it requires no synchronization at all. Secondly, operations `add`, `query` and `remove` require no synchronization at all. The only problem is the `sample` operation on average requires $S/(n-1)$ retries to succeed, where n is the number of elements in the set, assuming no concurrently executing writes. Each random choice can be thought of as a Bernoulli trial that succeeds if and only if the selected entry e is both in the array and is not i itself. There are exactly $n-1$ such entries in the set. The operation has a success probability of $(n-1)/S$. The expectation of said Bernoulli distribution is $S/(n-1)$.

At the initial stages of up-scaling, a thief can take a large number of trials to find a victim that is awake. This in theory causes our algorithm to respond slowly to emerging instantaneous parallelism. However, according to our experience, this is not a big problem when S is on the scale of hundreds because each trial involves very little effort.

SNZI Tree Random Set In this paragraph, we will present an algorithm for implementing data structure 2 with good theoretical guarantees. Our solutions consist of a tree of SNZI nodes (similar to SNZI objects s [13]), where every element in $\mathcal{S}$ corresponds to some different leaf SNZI node in the tree. In other words, there exists an injection from $\mathcal{S}$ to the set of leaf nodes in the SNZI tree. Every leaf node maintains a binary state according to whether its corresponding element, if exists, is in the set.

Definition 4. *A leaf node is said to be present if and only if its corresponding element, if exists, is in the set. A non-leaf node is said to be present if and only if any of its children are present.*

The idea of presence is similar to the idea of *Surplus* in [13]. We immediately see a non-leaf node is present if and only if one of the leaves in its subtree is present. Every SNZI nodes will support three operations: `Enter`, `Depart` and `Sample`. The first two operations announce (or cancels) the presence of the particular node. They are implemented in terms of operations of the parent node, if necessary. The third operation samples a present leaf node from that node's subtree. It's implemented using `Sample` operations of its children.

This algorithm does not impose any constraint on the shape of the tree other than there must be at lease S leaves. In fact, the optimal shape of the tree is usually machine-dependent. In general, taller trees further reduces the contention at each node, at the cost of decreasing the efficiency of `sample` operation. On the other hand, trees with a small height necessarily have more children at each node, which increases the contention. The optimal balance is clearly machine-dependent.

To efficiently implement those operations, every non-leaf node in the subtree will keep track of all of its present children. This can be easily achieved through an array-bimap based concurrent random set implementation introduced earlier. Therefore, the data structure for a single leaf SNZI node contains the following data fields:

`parent`	Parent SNZI node. `None` for the root node.
`present`	Boolean. `True` if and only if the leaf is present.
`elem`	The element associated with the node. `None` if the leaf is not associated with any element.

For a non-leaf node with k children:

`parent`	Parent SNZI node. `None` for the root node.
`set`	An array-bimap based concurrent random set of size k.

Algorithm 5 describes the operations on SNZI nodes.

Then the random set can be trivially implemented using SNZI tree. Our random set will keep track of the root node and the set of leaf nodes that corresponds to an element through the following data fields:

`root`	Root SNZI node. `None` for the root node.
`leaves[n]`	An array mapping each element to a leaf SNZI node.

Operations are now straight forward:

`new(S)`	Initialize an empty SNZI tree with a suitable structure (discussed in the following paragraphs).
`add(rs, i)`	Invoke `enterLeaf`(rs.`leaves[`i`]`).
`query(rs, i)`	Returns rs.`leaves[`i`].present`.
`remove(rs, i)`	Invoke `departLeaf`(rs.`leaves[`i`]`).
`sample(rs)`	Invoke `sampleNode`(rs.`root`).

Algorithm 5: Operations for SNZI node

```
fun enterLeaf(node):
    atomic
        node.present ← True;
        enterNonleaf(node.parent, node);
    end
end
fun enterNonleaf(node, child):
    atomic
        cnt ← add(node.set, child);
        if cnt = 0 and node.parent ≠ None then
            enterNonleaf(node.parent, node);
        end
    end
end
fun departLeaf(node):
    atomic
        node.present ← False;
        departNonleaf(node.parent, node);
    end
end
fun departNonleaf(node, child):
    cnt ← remove(node.set, child);
    if cnt = 1 and node.parent ≠ None then
        departNonleaf(node.parent, node);
    end
end
fun sampleNode(root):
    node ← root;
    while node isNot LEAF do
        node ← rsSample(node.set);
        // Retry in case of race condition
        if node = None then node ← root;
    end
    return node.elem;
end
```

Similar to the original SNZI algorithm, this design tries to reduce contention for write access by having child nodes filter access to parent nodes. Specifically, only accesses that changed the presence status of the current node are propagated up-wards, providing very fast `add` and `remove` operation. This ensures low contention at every non-leaf node, allowing us to choose array-bimap to implement concurrent random set at each non-leaf node. On the other hand, `sample` operation always takes time proportional to the height of the entire tree to complete. In general, the wider the tree, the higher the contention at each node, the faster the `sample` operation.

We briefly discuss the correctness of this algorithm. Because we are in a dedicated environment, atomicity is achieved by protecting each node with a lock. First of all, we ascribe `sample` operation with a relaxed specification. Then `add` operation must guarantee that the added element is observable by all future `sample` operations. This is why line 11 has to be protected by synchronization primitives: concurrent calls to `enterNode` to the same node may not return unless they are certain the change of presence, if required, has successfully propagated to the root. On the other hand, the `departNode` operation is more straight forward. Simply removing the current node from its parent's presence set prevents the node from ever being returned by future `sample` operations until it is added back.

Lack of synchronization between `sampleNode` and `departNode` introduces another race condition. A sample operation may choose a particular child from its parent. In the meantime the child is executing a `departNode` on its parent. The child is no longer present so that its presence set is empty, and trying to sample from the empty presence set will fail. We take care of this situation by backing off and simply restart the sampling process. We believe it is not a significant issue because when this happens, it means our algorithm has made progress in successfully putting some processor to sleep. This can happen at most a couple of hundreds of times before almost all processors have been put to sleep.

One particular problem with this design is that items in the set are not necessarily sampled uniformly. However, this problem is only significant in extreme cases where there are very few active processors. In the experiments we conducted in Chapter 4, we are unable to observe any problem. On the other hand, the logarithm complexity of the `sample` operation in the case of high instantaneous parallelism is somewhat unsatisfying. This can be partially mitigated by first performing k times random samples among all leaves before invoking `sampleNode`. This will allow us to avoid logarithm complexity when parallelism is high. Specifically, we may modify our `sample` operation according to Algorithm 6. With this algorithm, when parallelism is plenty, initial random choices have a high success probability. When the parallelism is low, the steal is guaranteed to succeed with at most $k + h$ random samples for a SNZI tree of height h.

Algorithm 6: `sample` operation with uniform samples.

```
1 fun sample(rs):
      // Perform $k$ times random trials.
2     for $i = 0$ to $k$ do
          // A random number $0 \leq i < S$
3         $t \leftarrow$ rand$(0, S)$;
4         if $rs$.leaves$[t]$.present then
5             return $rs$.leaves$[t]$.elem;
6         end
7     end
8     return sampleNonleaf($rs$.root);
9 end
```

Chapter 4

Experiments and Results

4.1 Introduction

In this section, we will evaluate our implementation of the elastic work-stealing scheduler as described in Chapter 3.

We will test our implementation against an existing implementation [21] of a traditional work-stealing algorithm, written by Daniel Spoonhower.

Our performance benchmarks come from mainly two sources: classical Cilk programs, the more recent problem-based benchmark suite (PBBS) by Blelloch et al [5]. The benchmark suite consists of highly parallel programs, and our goal is to show that our algorithm is as performant as a traditional work-stealing algorithm. The benchmarks we chose are listed in Table 4.1.

We will will test out result on the following benchmarks:

Our benchmarking platform contains four Intel(R) Xeon(R) CPU E7-8867 v4 CPUs, and each contains 18 cores and 36 hardware threads (through hyperthreading). The processor runs at 2.40 GHz base frequency and is capable of dynamic frequency scaling. The maximum possible frequency is 3.30 GHz. The test platforms run on Ubuntu 16.04 SMP, running Linux kernel version 4.10.0.

Benchmark	Description
`fib`	Compute Fibonacci number using the exponential algorithm. This benchmark is highly parallel and regular. Moreover, the performance of this benchmark is largely independent of the memory behavior of the scheduler. Input to the Fibonacci program is 42.
`mergesort`	A parallel merge sort program running on an input size of 10 million. This algorithm is a direct port of the *cilksort* program to SML.
`samplesort`	A cache efficient parallel sample sort program running on an input size of 10 million elements.
`histogram`	A histogram program based on an concurrent hash table. The performance of this program relies on the locality of memory accesses. Input size is 50 million elements.
`prime`	A parallel Sieve of Eratosthenes, expressed succinctly in terms of delayed sequences. This program looks for all primes within 50 million.
`mcs`	A parallel maximum sub-array algorithm for one-dimensional array using a naive search algorithm. The input array size is 200 million real numbers.
`bfs`	A parallel graph breadth-first search algorithm. Parallelism available relies heavily on the shape of the graph. We will run the benchmark with a randomly generated sparse graph (`bfssparse`) and dense graph (`bfsdense`). The sparse graph contains 100 million nodes with 500 million edges. The dense graph have 5×10^5 (0.5 million) nodes and 5×10^8 (500 million) edges.

Table 4.1: Benchmarks to evaluate the runtime perspective of our elastic work stealing algorithm.

4.2 Benchmarking Highly Parallel Computations

The benchmarking is conducted in the following manner. First, we will run a known best possible sequential algorithm for each problem. Then for each problem, we will run both the Spoonhower scheduler and our elastic scheduler, with 2, 4, 6, 8, 12, 18, 24, 32, 48, 64, and 72 cores. For each run, we will run the same algorithm on identical input 25 iteration consecutively. We will measure both the average runtime (wall-clock time) and average total CPU time for each problem. We measure the wall-clock runtime through `gettimeofday` system call, which provides microsecond accuracy on our platform. For the runtime, we will treat the first five runs as warm-ups, and take the average over the remaining 20 runs. We use GNU `time` utility available on most Linux systems to measure total work. It generates output in the following form:

```
113.32user 8.05system 1:00.75elapsed 199%CPU (...)
0inputs+0outputs (0major+27794minor)pagefaults 0swaps
```

Important metrics involves `user` and `system`.

user Measures **total** CPU time spent **in user space** executing the program, in seconds.

system Measures **total** CPU time spent **in kernel space** executing code on behalf of the program, e.g., due to system calls, in seconds.

We will sum up both components and use the sum as the total work performed by all processors. We will then divide this number by 25 to obtain the work performed in each iteration.

The structure of the SNZI random set is decided using the following algorithm. First, each level of the tree will have identical arity. Then we decide the arity for each level by decomposing the total number of processor P. For example, for $P = 24$, first obtain its prime decomposition $P = 24 = 2^3 \times 3$. Rewrite the power of 2 into a power of 4 with a possible extra multiplier of 2. In our case $P = 2^3 \times 3 = 4^1 \times 2 \times 3$. Order the product by descending base and assign the arity top to bottom in the order of the base. In our case, then the first layer has 4 children, the second layer has 3 children, and the last layer has 2 children. This intuition behind this algorithm is to reduce the height of the tree as much as possible within the range of tolerable contention.

Through Figure 4.1 to Figure 4.4, we will compare both the runtime speedup and relative total work for each problem. The speedup is computed by dividing sequential

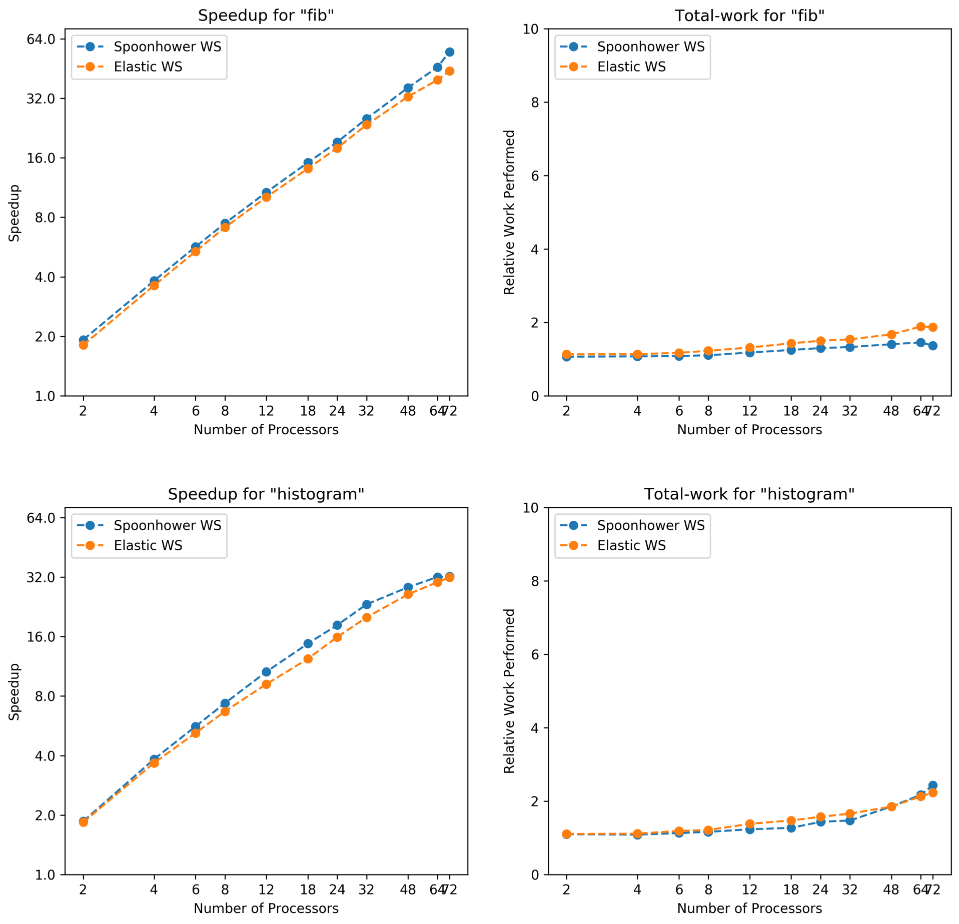


Figure 4.1: Comparison of runtime speedup and total work between the spoonhower scheduler and the elastic work-stealing scheduler on benchmarks problems `fib` and `histogram`

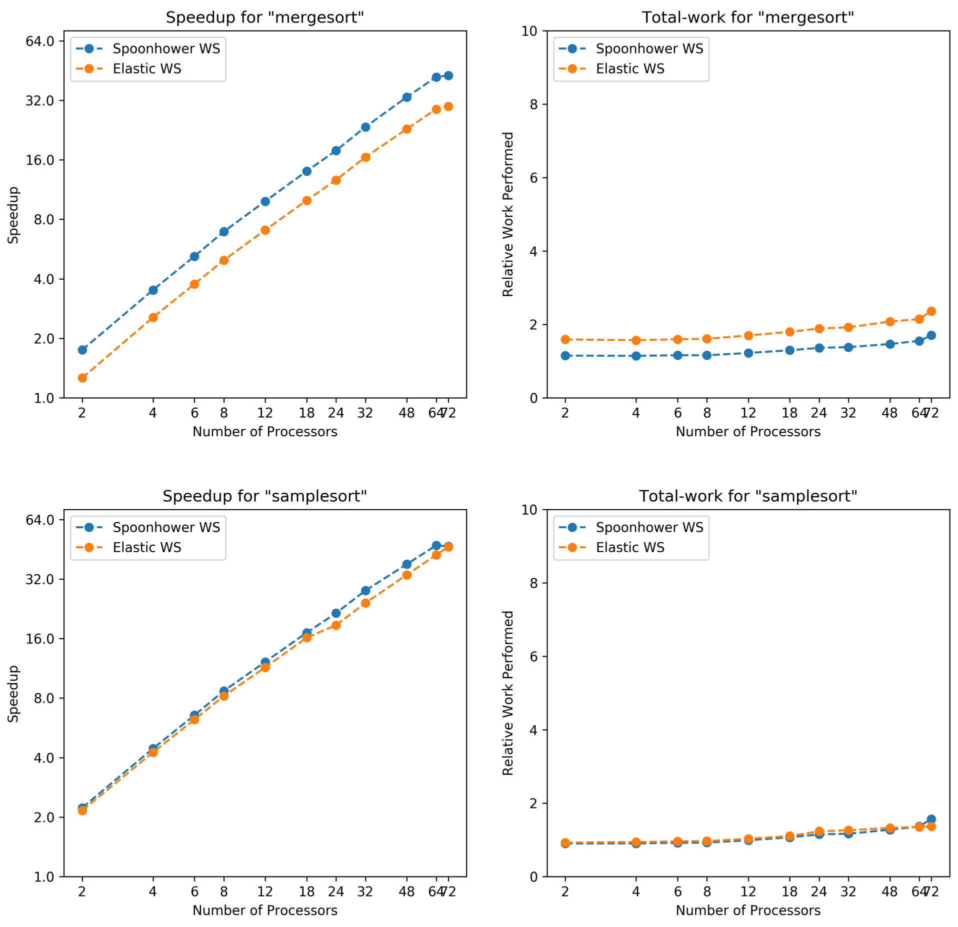


Figure 4.2: Comparison of runtime speedup and total work between the spoonhower scheduler and the elastic work-stealing scheduler on benchmark problems `mergesort` and `samplesort`.

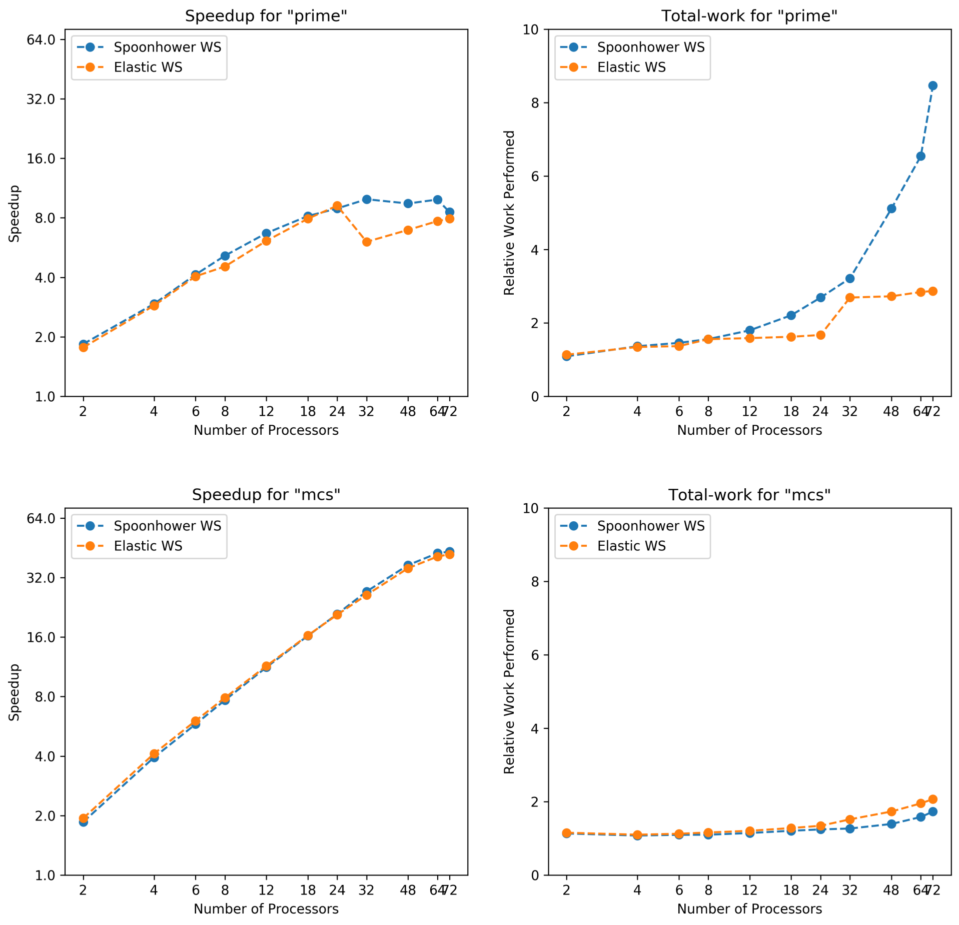


Figure 4.3: Comparison of runtime speedup and total work between the spoonhower scheduler and the elastic work-stealing scheduler on benchmark problems `prime` and `mcs`

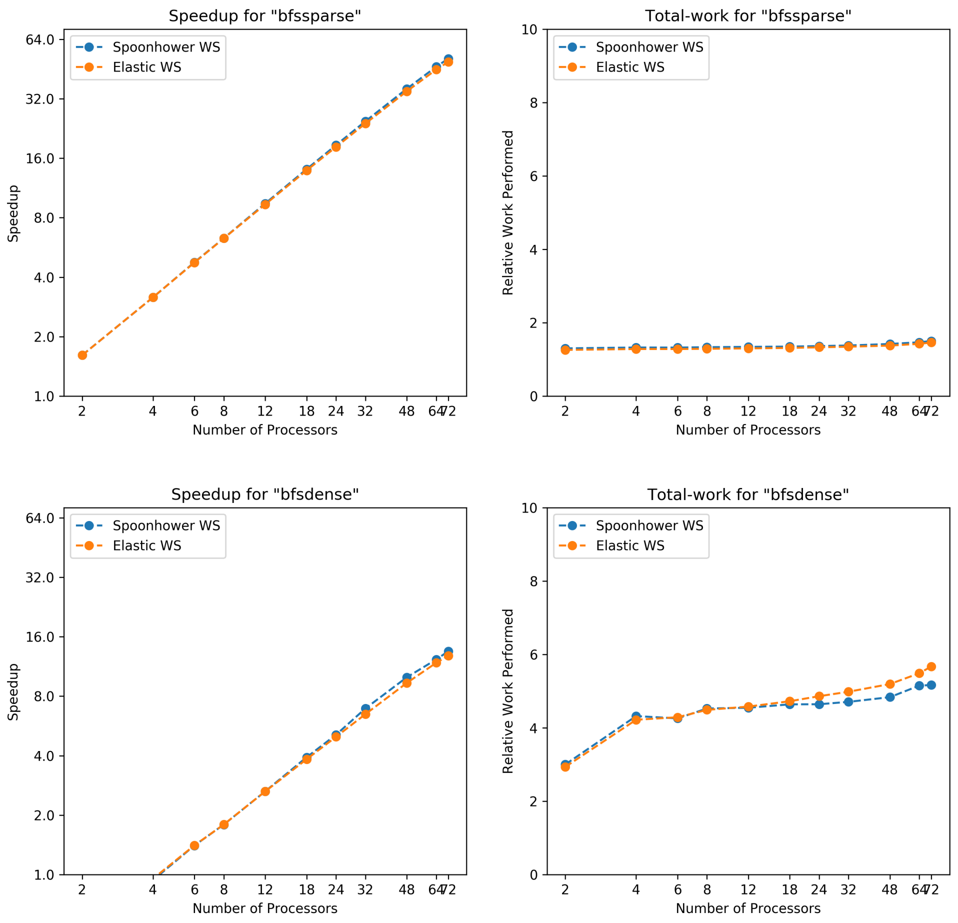


Figure 4.4: Comparison of runtime speedup and total work between the spoonhower scheduler and the elastic work-stealing scheduler on benchmark problems `bfssparese` and `bfsdense`

.

runtime by parallel runtime. The relative total work is computed by dividing the total work of the parallel run by the total work of the sequential run.

The data shows our scheduler achieves almost identical speedup compared with a traditional work-stealing scheduler. Part of the reason is that most benchmarks in the suite are highly parallel, and their instantaneous parallelism is almost always greater than the available processors in the system. On the other hand, our scheduler performs slightly more work compared with a standard work-stealing scheduler. This is mainly because each steal in the elastic work-stealing scheduler is slightly more expensive due to the need for synchronizations to guarantee correctness. The only exception here being the `prime` benchmark problem, which will be studied more closely in the next section.

4.3 Elasticity of the Elastic Work-Stealing Scheduler

In this section, we discuss the elasticity of the elastic work-stealing algorithm through the lens of the `prime` benchmark problem. In particular, we want to show that our work-stealing scheduler is capable of swiftly responding to varying parallelism, and it achieves it by performing only a constant factor extra work. Towards this goal, we built an event profiler into the scheduler. The event profiler aims to keep track of 1) the instantaneous parallelism in the system, measured by the number of ready threads in the system 2) The number of processors awake and 3) the number of processors performing actual computation. The profiler captures the following set of events:

`EventFork` A binary `fork` is executed and one extra task is spawned as a result. As a result, the instantaneous parallelism increase by 1.

`EventComplete` A piece of task has completed. As a result, the instantaneous parallelism drop by 1.

`EventSleep` A processor sleeps. The number of active processors decrease by 1.

`EventWakeup` A processor is waken up by one of its peers. The number of active processors increase by 1.

`EventStartStealing` A processor transitions into work-stealing. The number of processors executing user computation decrease by 1.

`EventObtainWork` A processor successfully obtain work through work-stealing. The number of processors executing user computation increase by 1.

The profiler will collect those events independently for every processor during the scheduling. At the end of the scheduling, the profiler combines and dumps those events into a file. The analysis of the log file is performed offline. The following results are obtained by running the `prime` benchmark program with 72 processors. In Figure 4.5, the `tasks` curve shows the number of tasks in the system. The `awake` curve shows the number of active processors in the system, and the `busy` curve shows the number of processors working on actual computation. In the second subfigure, we trimmed the Y-Axis by 100 because we are mostly interested in the behavior of our algorithm in the face of low instantaneous parallelism.

As one can see, the `prime` benchmark suite is particularly interesting for our purpose for two reasons: 1) Throughout the computation, the instantaneous parallelism has very high variance. It touches one even when the algorithm is still in "highly parallel phase", for instance, from 50ms to 100 ms. 2) From 100 ms to 300ms, the algorithm the instantaneous parallelism exposed by the algorithm is not sufficient to keep all processors busy. Those features are observable in Figure 4.6, where we have isolated the 50ms to 100ms section and 150ms to 200ms section.

We further isolate 70ms to 80ms, 110ms to 120ms, and 112ms to 116ms section in Figure 4.7. Figure 4.7 provides evidence for our claims. The scheduler scales-up and scales-down the processor utilization in response to increasing instantaneous parallelism in hundreds of microseconds. Moreover, our scheduler is work efficient because the number of awake processors is almost always a smaller constant larger than the number of processors executing actual computations. In other words, the work we waste is constant compared with the total work of the computation, and therefore it is empirically work-efficient.

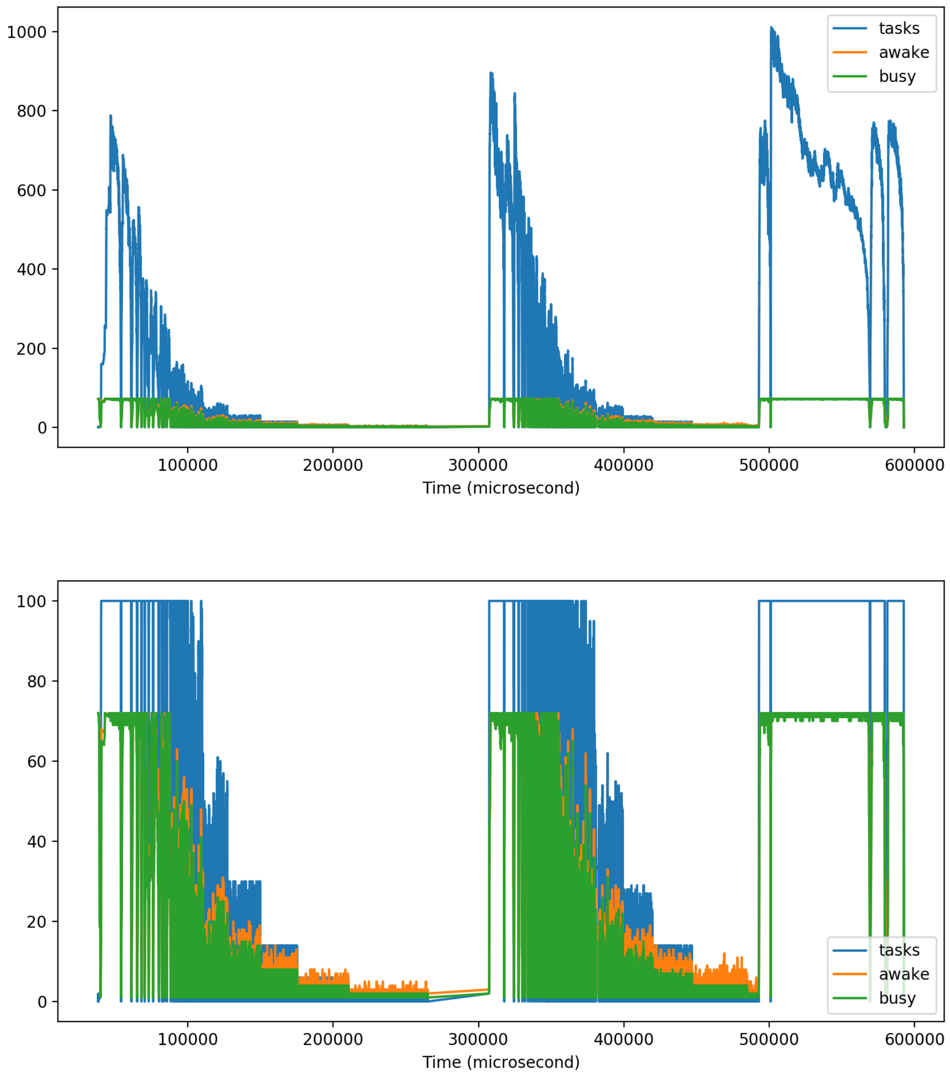
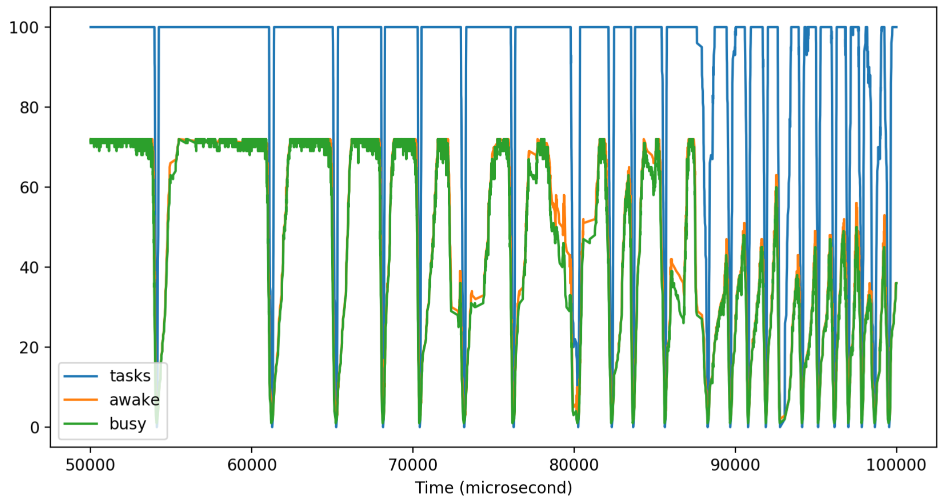


Figure 4.5: Profiling for the `prime` benchmark problem. The `tasks` curve shows the number of tasks in the system. The `awake` curve shows the number of active processors in the system, and the `busy` curve shows the number of processors working on actual computation. In the second subfigure, we trimmed the Y-Axis by 100 because we are mostly interested in the behavior of our algorithm in the face of low instantaneous parallelism.



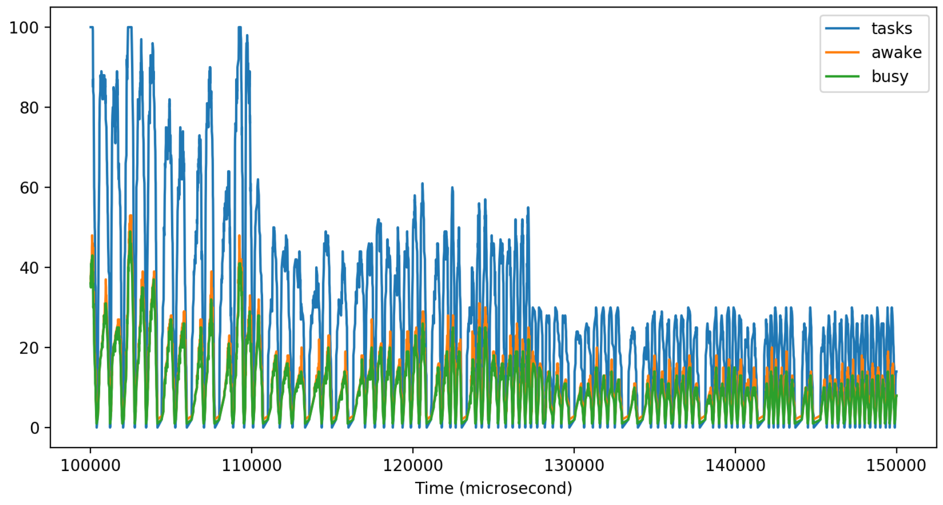


Figure 4.6: Profiling for the `prime` benchmark problem from 50ms to 100ms and 150ms to 200ms. Instantaneous parallelism varies dramatically and exhibits burstiness in both sections. From 150ms to 200ms, the computation has limited parallelism.

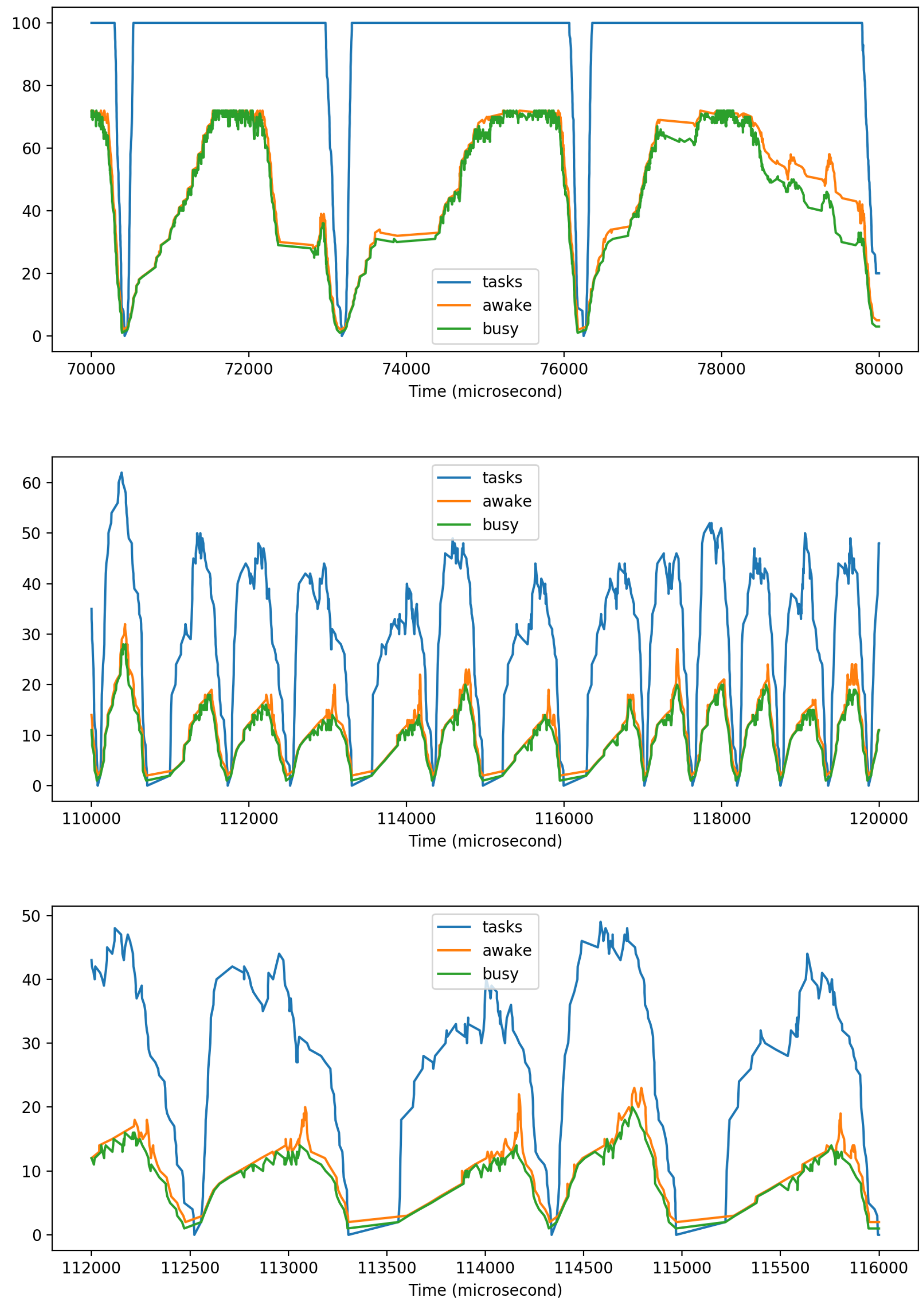


Figure 4.7: Profiling for the `prime` benchmark shows that the elastic work-stealing algorithm responds to varying instantaneous parallelism swiftly, and is work-efficient.

Chapter 5

Conclusions and Future Works

This thesis explores the idea of work-efficient schedulers. In order to achieve work-efficiency, this work identifies elasticity as a critical component in designing work-efficient schedulers. This work presents two work-efficient elastic schedulers for the dedicated environment: the offline $\alpha|\beta$-elastic greedy scheduler and the elastic work-stealing online scheduler. To formulate and analyze the offline scheduler, we start by defining a P-processor elastic scheduling model capable of considering offline elastic scheduling. We show that the $\alpha|\beta$-elastic greedy scheduler, being a particular case of the more general $\alpha|\beta$-elastic scheduler, is provably work-efficient and performant. This work further presents an online elastic work-stealing algorithm. The online work-stealing algorithm approximates the offline scheduler by utilizing the lifeline forest data structure to respond to varying instantaneous parallelism actively. We finally implemented the elastic work-stealing algorithm and showed that it is as performant as a traditional work-stealing algorithm on highly parallel tasks, and performs much less work on programs with limited parallelism.

We propose the following directions for future research:

- Extending the $\alpha|\beta$-elastic scheduling algorithm to the adaptive environment. This extension is not trivial because we will potentially need to extend the notion of $\alpha|\beta$-elasticity to the adaptive environment. In particular, the elasticity that a system can provide not only depends on the total number of workers but the number of workers that are currently active, because workers are in the end entities that respond to varying instantaneous parallelism. In an adaptive environment, our model will have to take this effect into account.

- Consider allowing the thief to steal half of the work queue instead of just one. *Steal-half* strategy has already been studied in the traditional work-stealing context [16].

The authors argue that in some cases stealing half of the deque provides better performance due to better stability and load balancing. Improved load balancing is especially interesting for our purpose because a balanced load ensures the recently joined processors can easily find work, and in turn, enables more processors, as long as there exists sufficient parallelism in the system. This further ensures an exponential rate scale-up in the face of increasing instantaneous parallelism.

- We may also investigate whether it's beneficial to implement the elastic work-stealing algorithm based on a private-deques [1]. With the lifeline forest meditating the communications between processors, a private deque implementation where processes explicitly communicate with each other for load balancing is quite approachable. A private deque implementation could significantly simplify synchronization and therefore improve performance. Moreover, it works well with the previously mentioned steal-half stealing strategy.

Bibliography

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 219–228. ACM, 2013. 1.1, 5

[2] Kunal Agrawal, Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–7. IEEE, 2007. 1.3

[3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 112–120. ACM, 2007. 1.3

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001. 1.1, 1.2, 1.3

[5] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In Ramanujam and Sadayappan [18], pages 181–192. 4.1

[6] Guy E. Blelloch, Phillip B. Gibbons, Girija J. Narlikar, and Yossi Matias. Space-efficient scheduling of parallelism with synchronization variables. In Charles E. Leiserson and David E. Culler, editors, *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '97, Newport, RI, USA, June 23-25, 1997*, pages 12–23. ACM, 1997. 1.1, 1.2, 1.2

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996. 1.1, 29

[8] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998. 1.2

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 1.1, 3.1

[10] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). In Mary K. Vernon, Garth Gibson, Guy Latouche, and Scott T. Leutenegger, editors, *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, SIGMETRICS '98 / PERFORMANCE '98, Madison, Wisconsin, USA, June 22-26, 1998*, pages 266–267. ACM, 1998. 1.1

[11] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974. 1.2

[12] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Computers*, 38(3):408–423, 1989. 1.2

[13] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: scalable nonzero indicators. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 13–22. ACM, 2007. 1.4, 3.3.2, 3.3.2, 10, 10

[14] Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 119–130. ACM, 2008. 1.1

[15] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223. ACM, 1998. 1.1

[16] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 280–289. ACM, 2002. 5

[17] Doug Lea. A java fork/join framework. In Dennis Gannon and Piyush Mehrotra, editors, *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, pages 36–43. ACM, 2000. 1.1

[18] J. Ramanujam and P. Sadayappan, editors. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. ACM, 2012. 5

[19] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. 1.1

[20] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar B. Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 201–212, 2011. 3.2

[21] Daniel Spoonhower, Guy E Blelloch, and Robert Harper. *Scheduling Deterministic Parallel Programs*. PhD thesis, Citeseer, 2009. 4.1

[22] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In Ramanujam and Sadayappan [18], pages 267–276. 1.1