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.

Copyright © 2019 Yue Yao
Keywords: Work Efficient Scheduler, Work Efficient Work Stealing Scheduler, 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 implement this algorithm then evaluate its performance and work efficiency by comparing it against existing implementations of traditional work-stealing schedulers. Results show that 1) for highly parallel computations, our parsimonious scheduler is comparable to classic work stealing in its speedup; 2) for computations where parallelism is more limited, our parsimonious scheduler performs considerably less work.
Acknowledgments

...
Contents

1 Introduction 1
  1.1 Overview .................................................. 1
  1.2 Multithreaded Computations as Computation Graphs .... 4
  1.3 Related Works .............................................. 7
  1.4 Contributions ............................................... 8

2 Work-Efficient Offline Scheduler 11
  2.1 An Offline Scheduling Game ............................... 11
    2.1.1 Greedy Schedulers in a Scheduling Game ............ 14
  2.2 $\alpha/\beta$-Elastic Greedy Scheduler ...................... 15
  2.3 Analysis of the $\alpha/\beta$-Elastic Greedy Scheduler .... 16
    2.3.1 Runtime Bound of $\alpha/\beta$-Elastic Greedy Scheduler 17
    2.3.2 Total-Work Bound of the $\alpha/\beta$-Elastic Greedy Scheduler 20

3 Elastic Working-Stealing Scheduler 23
  3.1 Introduction .............................................. 23
  3.2 Elastic Work-Stealing with Lifeline Forest ............... 24
  3.3 Lifeline Forest and Concurrent Random Set .............. 29
    3.3.1 Lifeline Forest .................................. 30
    3.3.2 SZNI Concurrent Random Set ......................... 35

4 Experiments and Results 45
List of Figures

1.1 An exemplary computation dag \( G \) consisting 6 thread \( \Gamma_1 \ldots \Gamma_6 \) and 28 tasks \( v_1 \ldots v_{28} \). \( v_0 \) is the root node and \( \Gamma_1 \) is the initial thread. Computation starts with task \( v_0 \) in \( \Gamma_1 \) and terminates with \( v_7 \) in \( \Gamma_1 \). For this graph, \( T_1 = 29 \), and the span \( T_\infty = 14 \).
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A 4-processor schedule of the exemplary computation dag in Figure 1.1.</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>The schedule takes exactly $T_\infty$ steps to complete, reaching the theoretical limit. Adding more processor cannot improve runtime.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Benchmarks to evaluate the runtime perspective of our elastic work stealing algorithm.</td>
<td>46</td>
</tr>
</tbody>
</table>
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 [17] and parallel ML [14].

One of the central challenges in designing and implementing fined grained parallel system is to efficiently schedule threads on to processors. The scheduler dynamically map threads onto processors, hoping to achieve linear overall 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. Schedulers can be studied in various setups.

We may classify schedulers in terms of the environments they operate under. If the number of processors made available to the scheduler stays constant throughout the entire scheduling, we refer this scenario as a dedicated environment. Conversely, if the the number of processors available varies, then the scheduler works in a multiprogrammed environment [4]. Scheduling in a multiprogrammed environments is also known as adaptive scheduling [2]. Previous works [4][5] proposed to model scheduling in an adaptive environment using a two-level scheduler model: there exists a task scheduler that schedules tasks on to a fixed set of workers (in some literature, processes), along with a poten-
tially adversarial job scheduler that schedules a subset the works onto the processors. In principle, the job scheduler corresponds to the operating systems’ scheduler, and workers correspond to operating system threads. The focus is on the design of task schedulers.

On the other hand, we may also classify schedulers in terms of the information they are allowed to take advantage of. Offline schedulers are omniscient in that have constant access to a global whole-system view and are allowed to make decisions accordingly without any overhead. In particular, offline scheduler schedulers are aware of the structural of the entire computation from the beginning and have control over the system state at any moment. Studying offline schedulers allows us to draw conclusions on 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 explicitly communications.

Previous research mainly focused on designing schedulers for better performance and (or) efficient space consumption. Those schedulers tries 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. Sources of those extra work includes overhead of scheduler code, communications between processors and most importantly, processors staying idle when there is not enough parallelism in the system at the moment. As a concrete example, the work-stealing scheduler described in will cause instruct a processor to keep “stealing” even when there is no viable target, burning CPU cycles while making no progress on the computation itself.

The goal of this work is to design and implement a new class of schedulers, called work-efficient schedulers. Intuitively, work efficient schedulers perform more work than the total work of the computation during the entire scheduling. These schedulers achieve work-efficiency by dynamically varying the amount of processors it utilizes and voluntarily give up processors if necessary. One of the potential applications for work-efficient scheduler is mobile computing. In the last decade we saw a rapidly increasing number of mobile devices equipped with multicore processors. An ever increasing amount of mobile applications rely on parallelism to deliver performance in an intermitted manner. Those application will have to rely on the runtime system to keep their CPU cycle consumption low when there is no parallelism available.

Define the maximum number of processors a computation can utilize at some moment during scheduling as the instantaneous parallelism of that moment. One justification for the work-inefficient design is that it allows the scheduler to respond quickly to fast increasing instantaneous parallelism. One of the key challenges in designing work-efficient scheduler is to maintain responsiveness while being work-efficient. This work takes re-
sponsiveness into account by defining elastic schedulers. A scheduler is called elastic if its processor utilization only increase or decrease within exponential rate in response to varying instantaneous parallelism. Elasticity puts constraint on how responsive schedulers can be. We show that elasticity is sufficient to guarantee both linear speedup and work-efficiency.

This work further propose an online scheduling algorithm that aims at approximating the offline work-efficient scheduler based on traditional work-stealing scheduler. Randomized work-stealing algorithm is very successful both in theory [9][4] and in practice [10][7][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 thread from the victim’s queue to its own 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 decrease 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 formulate the scheduling of a parallel computation as a scheduling game. Then we will propose an elastic offline scheduler. With the help of the scheduling game, we then prove that this algorithm is both work-efficient and performant by establishing both runtime bound and work bound for 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 implements those data-structures and discussed their correctness and performance.

- In Chapter 4 we implemented our elastic work-stealing scheduler and evaluate it by comparing with existing implementations of traditional work stealing algorithms. The results supports 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 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 computations, termed tasks, and edges represent dependencies between instructions. In the rest of this document, we use “tasks” and “nodes” interchangeably according to the context.

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

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

A computation must start with some node. 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 6 threads: each shaded region is a single thread. Since edges represent dependencies,
Table 1.1: A 4-processor schedule of the exemplary computation dag in Figure 1.1. The schedule takes exactly $T_\infty$ steps to complete, reaching the theoretical limit. Adding more processor cannot improve runtime.

nodes in a single thread forms a sequential computation. The thread of the root node is called the initial thread.

A $P$-processor schedule (or just schedule) of a computation graph is an assignment of node $n$ to processors $p_1 \ldots p_P$ for every time step $i$ that respects the dependencies that 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. Any valid schedule must respect dependencies.

At the beginning of $t$, the set of nodes whose dependencies are all satisfied is refer to the ready nodes of time $t$. In other words, ready nodes are all the nodes that can be scheduled without violating dependencies at time $t$. Given two different threads $\Gamma$ and $\Gamma'$, if the execution of some node $n$ of thread $\Gamma$ at time step $t$ puts node $n'$ of thread $\Gamma'$ 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 $\Gamma'$, we say $n$ spawned $n'$ (or equivalently $\Gamma'$). It is possible for a node to spawn multiple nodes. In our example, $v_1$ spawned $v_8$ and $v_{13}$ at time step 2. On the other hand, it’s also possible to have two nodes “simutaneously” 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 a smaller computation graphs at each time step. At 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 node in the graph. For this computation, \( T_1 = 29 \) since there are exactly 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 \( T_\infty \), is defined to be the length of the critical path in the graph. For this computation, there exists one unique critical path, consisting \( v_0, v_1, v_{13}, \ldots v_{16}, \ldots v_{25}, v_{28}, v_{20}, v_{21}, v_6, v_7 \). 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 scheduler at many ready nodes as possible at each step is referred to as 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 \( T_\infty \) span takes at most \( \frac{T_1}{P} + T_\infty \) 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 selects the “active” workers at each time step.

The model stated above is not sufficient for our purpose. In particular, the model say
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 \textit{processor utilization} of time \( t \). Those schedulers are called \textit{elastic schedulers}. This work considers a special class of elastic schedulers: an \( \alpha|\beta \)-elastic scheduler is a scheduler that only increases (or decreases) at an exponential rate, determined by the up-scaling factor \( \alpha \) and down-scaling \( \beta \). In other words, if an \( \alpha|\beta \)-elastic scheduler utilizes \( p_t \) processors at time step \( t \), then it may utilize at most \( \alpha p_t \) processors and at least \( p_t/\beta \) processors at time step \( t + 1 \).

1.3 Related Works

In \textit{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 \( T_1/P + O(T_\infty) \). Arora et. al. in \textit{Thread Scheduling for Multiprogrammed Multiprocessors} extended previous result to an online adaptive environment. Their work presented an non-blocking work-stealing scheduler implementation and proved that the work stealing scheduler completes any computation of in \( O(T_1/P_A + (P/P_A)T_\infty) \) runtime, where \( P_A \) is the average processor availability provided by the job scheduler. Both work did provide an analysis on the total work performed, but one can show that he online scheduler could perform \( O(T_1 + PT_\infty) \) total work in an dedicated environment. In particular, for almost sequential computations where \( T_1/T_\infty \) 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 \textit{Adaptive Scheduling with Parallelism Feedback}, Agrawal et. al presented the A-GREEDY offline scheduler for an adaptive environment. Their scheduler split the computations in to \textit{quantums} and explicitly communicate the number of processor it \textit{desires} to the job scheduler at the beginning of each quantum. This is known as the \textit{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 exponential rate, controlled by the responsiveness factor \( \rho \). 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 $T_1\rho T_1$ 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 Agrawal’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 don’t 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 a 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 a number of ways. Most importantly, our elastic scheduler is fully distributed. Every processor makes decisions based on their 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 an implementation and evaluated it on a real system. Our work has a few limitations. We only considered dedicated environment at this moment, and we didn’t provide a formal proof of runtime or work efficiency. Overcoming those limitation would be the next steps of our research.

## 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 game for analyzing offline elastic schedulers, suitable for both dedicated and adaptive environment.
• 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 $T_\infty$ span, the $\alpha|\beta$-elastic scheduler completes the computation in at most $\frac{T_1}{P} + T_\infty (1 + \log_\alpha \beta) + \log_\alpha P$ steps, performing at most $W \frac{\alpha \beta - 1}{\alpha (\beta - 1)}$ 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 key 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 our scheduler is both performant and work-efficient.
Chapter 2

Work-Efficient Offline Scheduler

2.1 An Offline Scheduling Game

In this section, we will begin by introducing a game theoretic model that enables us to account for varying processor utilization. This model will be based on a two-level scheduling model, commonly used for modeling adaptive scheduler. Even though this work focuses on scheduling in a dedicated environment, the game theoretic model itself is general enough to discuss offline adaptive scheduling, leaving up possibilities of future work.

Suppose there are \( P \) workers \( 1 \ldots P \) in the system. We may model a parallel schedule of a computation graph \( G \) onto \( P \) workers as a game played between the task scheduler (the player) and the job scheduler (the operating system scheduler, the opponent). In the rest of this document, we may use “task schedule” and “the player” interchangeably, and “job scheduler” and “the opponent” interchangeably.

In each round, the task scheduler assigns each work an intention, declaring whether this worker will be utilized, and how it will be utilized. The opponent chooses some of the workers and assigns them a processor, allowing the worker to carry out it’s intention. As a result, some tasks are executed, and those nodes are removed from the computation graph. The game terminates when all nodes are executed.

For each round, the game charges player some units of costs based on the resources consumed in this round. Resources could be wall-clock time or processor work. The goal of the player is to complete game and spend as little resource as possible.

We formally define the scheduling game between the task scheduler and the job scheduler as follows:
A scheduling game is a two-player game played on a sequence of computation graphs \( G_i \). In the first round, \( G_1 = G \) where \( G \) is the initial computation graph. The player and the opponent take turns to play the game. The goal of the player is to remove nodes from \( G \) with minimum total cost over all rounds. The cost is computed by a cost rule \( C \) chosen at the beginning of the game. The game terminates right before the opponent’s turn of round \( t \) if \( G_t \) becomes empty. Round \( t \) is referred to as the final round.

In each round, the player schedules a number of tasks onto \( P \) workers according to some scheduling policy \( X \). It does so by setting an intention \( I \) for each of the worker. An intention for a work is one of three options:

- A worker may decide to execute a task \( n \), denoted as \( E(n) \).
- A worker may choose to sleep for this round, denoted as \( \text{sleep} \).
- A worker may choose to stay idle (spinning), denoted as \( \text{spin} \).

The function from all workers to their intentions of round \( i \) is called the intention of round \( i \), denoted using \( I_i \). The player’s turn for round \( i \) ends when it decides on \( I_i \) according to its scheduling policy. A worker is said to be enabled if it’s assigned a non-sleep intention, otherwise it’s called disabled.

Different types of schedulers can be considered in this unified framework by substituting in different scheduling policies, making the name of a scheduler and its scheduling policy synonyms.

Then the job scheduler choose a non-empty subset (potentially equal) of workers and allow them to act out their intentions by conceptually assigning them a processor.

As the result if a task is executed by some processor as a result, then it’s removed from \( G_i \). Removing all executed tasks results in \( G_{i+1} \). Moreover, the game charges the player some cost \( c_i \) according previously agreed upon cost rule \( C \). Since the game terminates right before the opponents’ turn in the final round, the player incurs no cost in the final round.

The strategy of the player and the opponents both are subject to constraints. The task scheduler must respect data dependencies between tasks. Define the set of zero in-degree nodes \( \{ n_1, \ldots, n_r \} \) of \( G_i \) as the set of ready tasks at round \( i \), denoted as \( R_i \). The player may only execute nodes in \( R_i \) for round \( i \). In particular, \( r_i \triangleq |R_i| \) is termed the instantaneous parallelism of round \( i \).

The worker’s choice is subject to the following two constraints:
• It may only choose an worker with spin intention when all workers with executing intentions have been chosen.

• It must only choose an worker with sleep intention worker when all enabled workers have been chosen.

**Definition 1.** Define the number of workers chosen by the job scheduler at round $i$ as the processor availability of round $i$, denoted using $p_i$.

Those two constraints essentially requires the job scheduler to allocate processor availability first to executing workers, then to spinning ones, finally to sleeping ones. Those constraints are very important for the player because it allows the player to constraint the behavior of the opponent and as a result, avoid being over charged.

Next we will hope to show it’s possible to discuss both scheduling runtime and the total work of the scheduling by considering two different cost rules.

The runtime-cost rule $C_T$ charges the task scheduler 1 unit of cost at every round. Under the runtime cost rule, then intuitively define the runtime of a scheduling as the the total cost incurred by the task scheduler under rule $C_T$, which is just the number of rounds the entire scheduling took.

To define the cost rule to account for total work, we further make the following definitions:

**Definition 2.** For round $i$, the number of enabled workers is termed the worker utilization of round $i$, denoted using $\tilde{d}_i$. The number of chosen enabled is termed the allotment of the round $i$, denoted using $a_i$.

The work-cost rule $C_C$ charges the task scheduler $a_i$ unit(s) of cost each round. I.e., the incurs one unit of cost for each processor quanta it burned, no matter it’s spent on executing tasks or spinning. The we may define the total work of a $t$-round scheduling as the total cost incurred under rule $C_C$, namely $\sum_{i=1}^{t} a_i$.

$\tilde{d}_i$ reflects the number of processors the player intends to utilize. Notice that for all rounds $i$, $a_i \leq \tilde{d}_i$ because the job scheduler must prioritize over non-sleeping workers.

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

The distinction between a dedicated environment and a multiprogrammed environment can be viewed as two different classes of strategies for the job schedulers. If $p_i = P$
for all rounds \( i \), we say the scheduling game is played in a dedicated environment. In a dedicated environment, the opponent is benign: it will always select all workers at every round. This strategy will grant the player full control over how and when task are scheduled onto each processor. Conversely, the job scheduler may decide to vary the number of processors made available to the task scheduler at each round, resulting in an adaptive environment. In an adaptive environment, the job scheduler is potentially malign because it may choose to control how the computation graph is explored by the player.

In a dedicated environment, because all workers are chosen at each round, \( \tilde{d}_i = a_i \) for all rounds. Then the work-cost incurred by the player is just the sum of \( \tilde{d}_i \) over all rounds.

Because this work choose to focus on the dedicated environment, we will use “workers” and “processors” interchangeably in the rest of this document, and we will refer to the task scheduler as just scheduler.

### 2.1.1 Greedy Schedulers in a Scheduling Game

As a concrete example, let’s consider a traditional non-elastic greedy scheduler. A non-elastic greedy scheduler employs the following scheduling policy \( \lambda^\text{greedy}_i \):

**Scheduler Policy 1** (Non-elastic Greedy Scheduler). For every round \( i \), suppose \( R_i = \{r_1, \ldots, r_{r_i}\} \), assign tasks \( r_1, \ldots, r_{\min(P, r_i)} \) 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). The greedy scheduler completes the scheduling of a computation graph of \( T_1 \) work and \( T_\infty \) span in \( T_1/P + T_\infty \) steps given \( P \) processors.

**Proof.** We briefly go through the proof because the result is fairly well known. We classify rounds according to its instantaneous parallelism. For round \( i \), if \( r_i \geq P \), then \( P \) nodes are removed from \( G_i \). There can be at most \( T_1/P \) such rounds. 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 \( T_\infty \), there can be at most \( T_\infty \) such rounds. Combining both components, we see that there can be at most \( T_1/P + T_\infty \) rounds.

Since the scheduler is not elastic, we incur exactly \( P \) units of cost under \( C_C \) each round. Then the non-elastic greedy scheduler performs at most \( T_1 + PT_\infty \) work, which is \( PT_\infty \) extra work.

14
We may also consider the “elastic version” of the traditional greedy scheduler

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

The runtime of this scheduler is identical to the non-elastic greedy scheduler, however this scheduler is work efficient because this scheduler performs exactly $T_1$ work because we never put processors on idle. The problem with it is this scheduler requires unbounded *elasticity*, i.e., the scheduler will have to be able to respond to arbitrarily dramatic change in instantaneous parallelism in just one round. 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 $\alpha$ and $\beta$. 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 3 ($\alpha|\beta$-elasticity).** Given two real parameters $\alpha > 1$, $\beta > 1$, a scheduler is called $\alpha|\beta$-elastic if and only if, for all consecutive rounds $i$ and $i + 1$ where round $i + 1$ is not the final round, then $\frac{d_i}{\beta} \leq d_{i+1} \leq \alpha d_i$. In other word, processor utilization never increase faster than $\alpha$-fold or decrease faster than $\beta$-fold.

Now we present the $\alpha|\beta$-elastic greedy scheduling policy $X_E$:

**Scheduler Policy 3 ($\alpha|\beta$-elastic greedy scheduler).** *The $\alpha|\beta$-elastic greedy scheduler keeps track of a parameter $d_i \in \mathbb{R}$ for every round $s_i$. $d_i$ serves as the upper bound for processor utilization for round $i$. Let $d_0 \triangleq 1$. It determines the current round $d_i$ according to the following rules:*

- **Down-scale** *If $r < d_{i-1}$, then set $d_i = \max(r, \frac{d_{i-1}}{\beta})$.*
- **Satisfied** *If $r = d_{i-1}$, then keep $d_i = d_{i-1}$.*
- **Saturated** *If $r > d_{i-1} = P$, then keep $d_i = d_{i-1} = P$.*
Up-scale  If \( r > d_{i-1} \) and \( d_{i-1} < P \), then set \( d_i = \min(r, P, \alpha d_{i-1}) \).

For the final round, since \( r = 0 \), then the scheduler will simply carry out the down-scale rule.

Once the scheduler determined the \( d_i \) for current round, then it will enable exactly \( \tilde{d}_i = \lfloor d_i \rfloor \) processors and greedily schedule ready tasks on to those processors. The enabled processors without assigned tasks will simply spin.

Remark 1. Scheduling policy \( \mathcal{X}_E \) is constructed to make the following claims hold:

- For any round \( i \) other than the final round, \( 1 \leq d_i \leq P \).
- For each round \( \tilde{d}_i = \lfloor d_i \rfloor \).
- \( \mathcal{X}_E \) is \( \alpha|\beta \)-elastic.

\( \mathcal{X}_E \) utilizes the elasticity to dynamically to changing instantaneous parallelism. If the instantaneous parallelism is less than processors available in previous round the scale up rules increases the desired number of processors at a maximum rate of \( \alpha \). If the instantaneous parallelism is more than processors available in previous round, the scale down rules decreases the desired number of processors at a maximum rate of \( \beta \). 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 \( \mathcal{X}_E \)). \( \mathcal{X}_E \) completes any computation graph of \( T_1 \) work and \( T_\infty \) span with in at most \( \frac{T_1}{P} + T_\infty (1 + \log_\alpha \beta) \) rounds.

Theorem 2 (Total work bound of \( \mathcal{X}_E \)). In a dedicated environment \( \mathcal{X}_E \) completes any computation graph of \( T_1 \) work by performing at most \( T_1 \frac{\alpha \beta - 1}{\alpha (\beta - 1)} \) work.

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

To complete the proof, we further elaborate the rules of \( \mathcal{X}_E \) into the following six rules.
For scale-down rule, $r < d_{i-1}$. Let $d_i = \max(r, d_{i-1}/\beta)$. All $r$ ready tasks are scheduled.

a) If $r = d_i$. Parallelism drops within $\beta$ rate. $d_i < d_{i-1}/\beta$. All ready tasks are scheduled.

b) If $r < d_i$. Parallelism drops at a rate higher than $\beta$. $\lfloor d_i \rfloor - r$ processors are spinning. $d_i = d_{i-1}/\beta$. All ready tasks are scheduled.

c) $r = d_{i-1}$: Remains unchanged. All $r$ ready tasks are scheduled.

d) $r > d_{i-1} = P$: Remains unchanged.

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

e) $r = d_i$, Parallelism increases within $\alpha$ rate. All ready tasks are scheduled. $d_i > \alpha d_{i-1}$
f) $r > d_i$, Parallelism increases at a rate higher than $\alpha$. $d_i = \alpha d_{i-1}$. $r - \lfloor d_i \rfloor$ ready nodes are left for future rounds to execute.

Because $d_i \leq P$, 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 round $i$ is governed by rule any of the rules a), b), c) or e), then span of the remaining graph decrease by 1, because all zero in-degree nodes are excuted.

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 round in the scheduling game according to the scheduling rule applied to that round.

Definition 4. A round is a $r$-round if scheduling rule $r$ is applied for that round.

The proof proceeds by categorize the rounds according whether rule d) that governs the round. We bound the the positive and negative cases separately and combine the counts in the end.

Lemma 3. There are at most $T_i/P$-rounds (saturated rounds).
Proof. Each d)-round decrease the size of $G$ by $P$. Note that $|G| = T_1$ by definition, therefore there may be at most $T_1/P$ d)-rounds.

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

$$
\Phi(i) \triangleq T_i^\infty (1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i}
$$

for round $i$, where $T_i^\infty$ is span of graph $G_i$. Consider the consecutive difference:

$$
\Phi(i) - \Phi(i + 1) = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} \geq 0
$$

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 of $X_E$ for round $i$:

**Rule a) & b)** For rule a), $T_i^\infty - T_{i+1}^\infty = 1$, $d_i \geq d_{i-1}/\beta$. Then

$$
\Phi(i) - \Phi(i + 1) = (1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} \geq (1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} = 1
$$

**Rule c)** For rule c), $T_i^\infty - T_{i+1}^\infty = 1$, $d_i = d_{i-1}$

$$
\Phi(i) - \Phi(i + 1) = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} = 1 + \log_\alpha \beta > 1
$$

**Rule d)** For rule d), $d_i = d_{i-1}$, $T_i^\infty \geq T_{i+1}^\infty$. Then

$$
\Phi(i) - \Phi(i + 1) = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) \geq 0
$$

**Rule e)** For rule e), $T_i^\infty - T_{i+1}^\infty = 1$, $d_i > d_{i-1}$. Then

$$
\Phi(i) - \Phi(i + 1) = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} \geq (1 + \log_\alpha \beta) + \log_\alpha \frac{d_i}{d_{i-1}} = 1
$$

**Rule f)** For rule f), $T_i^\infty - T_{i+1}^\infty \geq 0$, $d_i = ad_{i-1}$. Then

$$
\Phi(i) - \Phi(i + 1) = (T_i^\infty - T_{i+1}^\infty)(1 + \log_\alpha \beta) - \log_\alpha \frac{d_i - 1}{d_i} \geq 0 - \log_\alpha \frac{d_i - 1}{ad_i} = 1
$$
We have analyzed all cases.

The analysis on consecutive difference allows us to bound the number of non-d)-rounds by the potential function:

**Lemma 5.** There are at most \( T_\infty (1 + \log_\alpha \beta) + \log_\alpha P \) non-d)-rounds.

**Proof.** For the first round, \( T_\infty^1 = T_\infty \) and \( d_0 = 1 \) by definition.

\[
\Phi(1) = T_\infty^1 (1 + \log_\alpha \beta) - \log_\alpha d_0 = T_\infty (1 + \log_\alpha \beta)
\]

Suppose the computation finishes after \( t \) rounds. As noted in Remark \[\text{[1]}\] \( d_{t-1} \leq P \). At the end of the computation, \( T_\infty^t = 0 \). Then

\[
\Phi(t) = T_\infty^t (1 + \log_\alpha \beta) - \log_\alpha d_{t-1} \leq -\log_\alpha P
\]

Then the potential difference is:

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

Since each non-d)-round decreases the potential by at least 1, and the potential decreases monotonically for all rounds, therefore they may be at most \( T_\infty (1 + \log_\alpha \beta) + \log_\alpha P \) non-d)-rounds. \( \square \)

The main result is now directly derivable:

**Proof.** Combining the results of lemma \[\text{[3]}\] and lemma \[\text{[5]}\] allows us to conclude theorem \[\text{[1]}\]. A round is either a d)-round or non-d)-round, therefore there are at most \( \frac{T_1}{P} + T_\infty (1 + \log_\alpha \beta) + \log_\alpha P \) rounds. This completes our proof for the performance bound. \( \square \)

Theorem \[\text{[1]}\] suggests our scheduler may take at most \( T_\infty \log_\alpha \beta + \log_\alpha P \) rounds compared to a greedy scheduler. We would like to provide some insight into the difference.

The bound contains the an additive term \( \log_\alpha P \). This term exists because we initialized the utilization \( d_0 = 1 \). Even if the computation is highly parallel, the scheduler still need \( \log_\alpha P \) rounds 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 $\beta$. This might be surprising at first glance. In fact, $\beta$ controls how susceptible our algorithm is to an adversarial computation graph may be. A large $\beta$ allows the processor utilization to drop quickly, which will take more rounds to ramp up again. A small $\beta$ 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:

$$\Phi(i) = T_i^1(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}d_{i-1}$$

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

$$\Phi(i) - \Phi(i+1) = (T_i^1 - T_{i+1}^1)(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

**Lemma 6.** The consecutive difference $\Phi(i) - \Phi(i+1) \geq d_i$ for all rounds $i$. i.e., $\Phi(i)$ decrease by at least $d_i$ for every rounds.

**Proof.** Proof by casing on all rules of $X_E$ for round $i$:

**Rule a)** $T_i^1 - T_{i+1}^1 = d_i$ and $d_{i-1} > d_i$. Then

$$\Phi(i) - \Phi(i+1) = (T_i^1 - T_{i+1}^1)(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$= d_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$> d_i(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + 0$$

$$> d_i(1 + 0) = d_i$$

**Rule b)** $d_i = d_{i-1}/\beta$, which is $d_{i-1} = \beta d_i$. $W_i - W_{i+1} > 0$. Then

$$\Phi(i) - \Phi(i+1) = (T_i^1 - T_{i+1}^1)(1 + \frac{\alpha - 1}{\alpha}\frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$= \frac{1}{\beta - 1}(\beta d_i - d_i) = d_i$$
Rule c) & d) In both cases, $T_i^1 - T_i^{i+1} = r = d_{i-1}$, and $d_i = d_{i-1}$ Then

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

Rule e) & f) In these cases $T_i^1 - T_i^{i+1} = d_i$ and $d_{i-1} < d_i \leq \alpha d_{i-1}$. Then

$$\Phi(i) - \Phi(i + 1) = (W_i - W_{i+1})(1 + \frac{\alpha - 1}{\alpha} \frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$= d_i (1 + \frac{\alpha - 1}{\alpha} \frac{1}{\beta - 1}) + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$= d_i + d_i \frac{\alpha - 1}{\alpha} \frac{1}{\beta - 1} + \frac{1}{\beta - 1}(d_{i-1} - d_i)$$

$$= d_i + (d_i \frac{\alpha - 1}{\alpha} + d_{i-1} - d_i) \frac{1}{\beta - 1}$$

$$= d_i + (d_{i-1} - d_i \frac{1}{\alpha}) \frac{1}{\beta - 1}$$

We have noted earlier, $d_i \leq \alpha d_{i-1}$. I.e., $d_i/\alpha \leq d_{i-1}$. Rewrite the result:

$$\Phi(i) - \Phi(i + 1) = d_i + (0) \frac{1}{\beta - 1} = d_i$$

We have analyzed all cases.

Now we are ready to prove Theorem 2.

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

$$\Phi(1) = T_1^1 (1 + \frac{\alpha - 1}{\alpha} \frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1} d_0 = T_1 (1 + \frac{\alpha - 1}{\alpha} \frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1}$$

$$\Phi(t) = T_t^1 (1 + \frac{\alpha - 1}{\alpha} \frac{\beta}{\beta - 1}) + \frac{\beta}{\beta - 1} d_{t-1} = 0 + \frac{\beta}{\beta - 1} d_{t-1} \geq \frac{\beta}{\beta - 1}$$

Then potential difference

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

21
On the other hand, by lemma 6,

\[ \Phi(i) - \Phi(t) = \sum_{j=1}^{t-1} (\Phi(j) - \Phi(j+1)) \geq \sum_{i=1}^{t-1} d_j \]

As Remark 1 has noted, \( \tilde{d}_i = \lfloor d_i \rfloor \). Then the total cost incurred under cost rule \( C_C \) is

\[ \sum_{i=1}^{t-1} a_i \leq \sum_{i=1}^{t-1} d_i \leq \Phi(i) - \Phi(t) = T_1(1 + \frac{\alpha - 1}{\alpha} \frac{1}{\beta - 1}) = T_1 \frac{\alpha \beta - 1}{\alpha(\beta - 1)} \]

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 \( \alpha \) and \( \beta \). 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 \( \alpha \). For sufficiently large \( \beta \), the constant is very small. This shows our algorithm is work-efficient.

Finally, decreasing \( \alpha \) also help 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. Here we borrow the offline world terminology “disable” and “enable”, meaning putting a processor to sleep and waking up a processor respectively. We further argue that such design achieve exponential rate up 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 $Q_p$. Processor pushes and pops tasks to their own work queue from 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 task from the bottom of its work queue. If the work queue is empty. The processor will try
December 2, 2019

DRAFT

to obtain task from other processor 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.

At 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 *theif*. The theif uniformly randomly choose another processor in the system. The chosen processor is termed the *victim*. The theif then attempts to pop the work queue of the victim from the top. If victim’s work queue is non-empty, then the operation succeeds and the theif starts working on the task it just obtained. This ends the work stealing. If the victim’s work queue is non-empty, then the theif 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 an 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 pretty sure that the instantaneous parallelism is low, and it should go to sleep.

- When a stealing processor encounter another stealing processor, it’s more work efficient to have the other processor 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 disable themselves and (or) enable other processors during work stealing. In the elastic work-stealing scheduler, when a processor becomes a *theif*. The theif uniformly randomly choose another *currently enabled* processor. The chosen processor is termed the *victim*. The theif then attempts to pop the work queue of the victim from the top as usual.

24
• If the victim’s work queue is empty and the victim is also stealing. Then the thief will try 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 succeeds, then it disables itself 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 obtained work. The thief enables 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 [18] of Saraswat et. al. In Saraswat’s work, they considered a fixed grid-shaped lifeline structure for a 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 forms a forest. In the next paragraph, we will introduce a new data structure called lifeline forest, that can be used to manage the communication structure between processors.

A lifeline forest is a forest (a number of trees) with a fixed number of nodes (called endpoints). Edges from one endpoint to another corresponds to a lifeline from former to the latter (the latter is responsible to send the message). Roots of the forest is the set of zero out-degree endpoints. Those endpoints are called independent endpoints. When a lifeline is formed, one of the independent node becomes a children 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...*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, and 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 lifeline tree in a straightforward 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 become 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 computation graph have a maximum out degree of two. This corresponds to the reality that many existing programming systems 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 for find another stealing processor is proportional to the number of thieves in the systems,
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 */
1 Set $n_0$ to be the root task, all other $n_p$ to None;
/* Scheduling loop for each processor $p$. */
2 while True do
3     while $n_p \neq \text{None}$ do
4         newTask ← Execute($n_p$);
5         if newTask \neq \text{None} then
6             pushBottom($Q_p$, $n_p$);
7             $n_p$ ← newTask;
8             continue;
9         end
10     end
11     $n_p$ ← popBottom($Q_p$);
12     end
13     while True do
14         shouldSleep ← False;
15         victim ← $L$.sample();
16         $n_p$ ← popTop($Q_{\text{victim}}$);
17         if $n_p \neq \text{None}$ then
18             $L$.signal($p$);
19         else
20             atomic
21             if $n_{\text{victim}} = \text{None}$ then
22                 // The victim is stealing or sleeping.
23                 shouldSleep ← $L$.attach($p$, $\text{victim}$);
24             end
25         end
26         if shouldSleep then
27             $L$.wait($p$) // Put myself to sleep.
28         end
29     end
30 end
a thief is very likely to find another thief. As the 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 exponential decrease. In other words, when the parallelism is low, the
lifeline forest “folds” at exponential rate.

• When the parallelism comes back, the algorithm start to “unfold” the lifeline forest.
Suppose there are plenty of parallelism in the system, then almost all steals succeeds
and further unfolds the lifeline forest. Notice that the unfolding processes is simply
the “reverse” process of formation of lifeline forest. Thus we should observe an
exponential increase in processor utilization.

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

The efficient work stealing algorithm will is $\text{safe}$ 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 $\text{victim}$ that results in an violation property 1. Now $p$ becomes an orphaned
sleeping processor that potentially will never be waken up (because $\text{victim}$ may always
have tasks to do). We effectively lose one processor in this case. Violating property 2
will, in the worst case, we lost all processors participating the loop (along with all of their
children).

Remark 3. The work efficient work stealing algorithm described in algorithm 7 is $\text{safe}$.

We argue that our algorithm is safe by examining each property separately.
Property 1 Consider the moment when $p$ attach a lifeline to $\textit{victim}$ on line 21 when it observes $n_{\textit{victim}} = \text{None}$ and the victim’s $Q_{\textit{victim}}$ 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’s before line 15 then later on it either continues stealing or removes 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 $n_{\textit{victim}}$ is $\text{None}$. We have examined all cases.

Property 2 Property 2 is automatically satisfied by the guarantees provided by the $\texttt{attach}$ 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 which endpoints that are independent. 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 $\text{True}$ if the item is in the set, $\text{False}$ 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 $\text{None}$ to signify failure.

Ideally, the concurrent random set supports concurrent $\texttt{add}$, $\texttt{remove}$ and $\texttt{sample}$ 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 allows 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. 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 concurrent random set are atomic. Atomicity guarantees can be implemented trivially using locks in most systems. This algorithm also make 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 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.

1 fun attach($L, p, v$):
   atomic
   if $p \neq v$ and $L.rs.exists(p)$ and $L.rs.exists(v)$ then
      append($L.nodes[v].lifelines, p$);
      $L.rs.remove(p)$;
      return True;
   end
   return False;
   end

2 fun signal($L, p$):
   atomic
   if not $L.rs.exists(p)$ 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

3 fun sample($L$):
   $L.rs.sample()$;
   end

4 fun wait($L, p$):
   atomic
   toWait \leftarrow False;
   if not $L.rs.exists(p)$ then
      toWait \leftarrow True;
      $L.nodes[p].waitCnt \leftarrow 1$;
   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 amount of processors blocked on \( n \).

4. No processor can be blocked on an independent endpoint.

5. For all endpoints, \( \text{sem} \) is always zero.

We can verify that the invariants are satisfied for a newly created lifeline forest. 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’s not in the set \( rs \), then the function also simply 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 on \( p \). 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 a independent endpoint. By definition of independence, its lifelines contains all endpoints that depends 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’s left to argue that the invariants are preserved. All previously dependent processors, which by invariant are previously removed from \( rs \) are added back as a result of \( rs.add \). There will be no processors blocked on for any \( v \) so that 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 process will eventually be blocked on \( p \) because by
invariant the semaphore is zero. By invariant \( \text{waitCnt} \) holds the number of processors currently blocked on \( p \). Because calling processor will ends up blocked on \( p \), we increase \( \text{waitCnt} \) by 1. We have verified all invariants and \( \text{wait} \)'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’s left to argue that operations in Algorithm 2 is linearizable. Unfortunately the it’s not the case, and it almost works. Here we present a case where Algorithm 2 fails to block a processor when there do 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 succeeds.
- 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 \( \text{wait} \).

This would happen because the newly established lifeline should see no processor being blocked on the lifeline. However, in Algorithm 2, a newly established lifeline falsely “inherits” the same \( L.waitcnt[n_1] \) from previous lifeline. Algorithm 3 solves this problem by versioning the semaphore and \( \text{waitcnt} \). Whenever a new lifeline is created, it increases the version number so that all other operations always operate on the latestest lifeline, if there exists one. To implement a lifeline forest with versioning requires us to maintain the following datafields for each endpoint:

\( v \) Version number of its own lifeline. Initialized to zero.

\( \text{sem}[v] \) 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.

1 fun attach($L$, $p$, $v$):
2     atomic
3         if $p \neq v$ and $L.rs.exists(p)$ and $L.rs.exists(v)$ then
4             ver ← $L.nodes[p].v + 1$;
5             sem_init($L.nodes[p].sem[ver]$, 0);
6             $L.nodes[v].waitCnt[ver]$ ← 0;
7             $L.nodes[v].v$ ← ver;
8             append($L.nodes[v].lifelines$, $p$);
9             $L.rs.remove(p)$;
10             return True;
11         end
12         return False;
13     end
14 end

15 fun signal($L$, $p$):
16     atomic
17         if not $L.rs.exists(p)$ then return;
18         ver ← $L.nodes[p].v$;
19         foreach $v$ in $L.nodes[p].lifelines$ do
20             $L.rs.add(v)$;
21             for $i = 1$ to $L.nodes[v].waitCnt[ver]$ do
22                 sem_up($L.nodes[v].sem[ver]$)
23             end
24         end
25         clear($L.nodes[p].lifelines$)
26     end
27 end

18 fun wait($L$, $p$):
19     atomic
20         ver ← $L.nodes[p].v$;
21         toWait ← False;
22         if not $L.rs.exists(p)$ then
23             toWait ← True;
24             $L.nodes[p].waitCnt[ver]$ += 1;
25         end
26     end
27     if toWait then sem_down($L.nodes[p].sem[ver]$);
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 claim 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. We proceed by demonstrating a linearization point for each of the operations. Clearly, linearization point of operations attach and signal is the exit point of the atomic region. For wait, the linearization point is right after line 36.

### 3.3.2 SZNI Concurrent Random Set

In this section we will discuss how to design SNZI concurrent random set. To remind the readers on what a concurrent random set is, here we reiterate the specifications on the operations it should support. A concurrent random set is a concurrent data structure supporting 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.

As stated before, elements in a concurrent random set must be drawn from a fixed, predetermined, integer indexable set $S$. The range of possible elements is termed the space of the elements, and the size of the space is denoted using $S$. Ideally an implementation of should provide fast and contention free access to the data structure, while maintaining 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 an practical implementation. To be formal:
Definition 5 (Relaxed specification for sample operation). An implementation of 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 uniformly select a victim for processors. 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.

- 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 will be acceptable.

- When instantaneous parallelism starts to vary, processors frequently goes to sleep or wake up depending on whether the parallelism is decreasing or increasing, it’s important for sample to provide up-to-date response. We want to ensure processors does not waste effort trying to steal from an already sleeping processor during down scaling, or trying to steal from a processor that haven’t been waken up yet during up scaling.

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 very 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 structure 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 $0 \leq p < \text{count}$, return elements[$p$].

---

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

1. fun rsRemove(rs, i):
2.     atomic
3.         rs.count ← rs.count − 1;
4.         l ← rs.locations[i];
5.         v ← rs.elements[rs.count];
6.         rs.locations[i] ← −1;
7.         rs.elements[l] ← v;
8.         rs.locations[v] ← l;
9.     end
10. end
For this simple design, operations add, query and remove all complete in constant time. Operation sample is expected to complete in constant number of retries in the case of relatively low contention because the probability of consecutive failure diminishes exponentially as number of trials increase. However operations add, query and remove must be synchronized to ensure correctness. In reality, the complexity of remove operation made it hard to implement synchronization without using a lock. 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 is a single point of contention when there exists when the algorithm is trying to scaling the processor utilization, which is not an ideal implementation for our purpose.

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 an 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 initial stages of up scaling, processors takes 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, as we will see in section 4.2 in practice 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 with good theoretical guarantees. Our solutions consist a tree of SNZI nodes (similar to SNZI objects s [13]), where every element in $S$ corresponds to some different leaf SNZI node in the tree. In other words, there exists an injection from $S$ to the set of leaf nodes in the SNZI tree. Every leaf node maintains a binary state according whether its corresponding element, if exists, is in the set.

**Definition 6.** 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 is 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 announces (or cancels) the presence the particular node. They are implemented in terms of operations of 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 least $S$ leaves. In fact, the optimal shape of the tree is usually machine dependent. In later paragraphs we will discuss how the shape of the tree affect various characteristic of this algorithm.

To efficiently implement those operations, every non-leaf node in the sub tree 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, 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** 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

1 fun enterLeaf(node):
  2    atomic
  3    node.present ← True;
  4    enterNonleaf(node.parent, node);
  5 end

6 end

7 fun enterNonleaf(node, child):
  8    atomic
  9    cnt ← add(node.set, child);
 10    if cnt = 0 and node.parent ≠ None then
 11       enterNonleaf(node.parent, node);
 12    end
 13 end

14 end

15 fun departLeaf(node):
  16    atomic
  17    node.present ← False;
  18    departNonleaf(node.parent, node);
  19 end

20 end

21 fun departNonleaf(node, child):
  22    cnt ← remove(node.set, child);
  23    if cnt = 1 and node.parent ≠ None then
  24       departNonleaf(node.parent, node);
  25    end
  26 end

27 fun sampleNode(root):
  28    node ← root;
  29    while node isNot LEAF do
  30       node ← rsSample(node.set);
  31       // Retry in case of race condition
  32       if node = None then node ← root;
  33    end
  34    return node.elem;
  35 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 current node is 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 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 correctness of this algorithm. Because we are in an dedicated environment, atomicity is achieved by protecting each node with a lock. First of all, we ascribe sample operation with an 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 exist, has successfully propagated to the root. On the other hand, the departNode operation is more straight forward. Simply removing 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 mean time 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 asleep.

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 when there is very few processors. In the experiments we conducted in section 4.2, we weren’t able to observe any problem. On the other hand, the logarithm complexity of the sample operation in the face of high instantaneous parallelism is somewhat unsatisfying. This can be partly 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 success with at most $k + h$ random samples for a SNZI tree of height $h$. 
Algorithm 6: sample operation with uniform samples.

```plaintext
fun sample(rs):
    // Perform k times random trials.
    for i = 0 to k do
        // A random number 0 ≤ i < S
        t ← rand(0, S);
        if rs.leaves[t].present then
            return rs.leaves[t].elem;
        end
    end
    return sampleNonleaf(rs.root);
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 [19] of 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] and a number of interesting parallel programs written by ?. Those benchmarks consists for highly parallel programs and the 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:

4.2 Experiments and Results
## Table 4.1: Benchmarks to evaluate the runtime perspective of our elastic work stealing algorithm.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fib</strong></td>
<td>Compute fibonacci number using the exponential algorithm. This benchmark is highly parallel and regular. Moreover performance of this benchmark does not rely heavily on memory behavior of the scheduler.</td>
</tr>
<tr>
<td><strong>mergesort</strong></td>
<td>A parallel merge sort program running on an input size of 10 million. This algorithm is a direct port of the <em>cilksort</em> program to SML.</td>
</tr>
<tr>
<td><strong>samplesort</strong></td>
<td>A cache efficient parallel sample sort program running on an input size of 10 million elements.</td>
</tr>
<tr>
<td><strong>histogram</strong></td>
<td>A histogram program based on an concurrent hash table. The performances of this program relies on locality of memory access.</td>
</tr>
<tr>
<td><strong>bfs</strong></td>
<td>A parallel graph breadth-first search algorithm. Parallelism available relies heavily on the shape of the graph.</td>
</tr>
<tr>
<td><strong>primes</strong></td>
<td>A parallel prime number sieve.</td>
</tr>
<tr>
<td><strong>mcs</strong></td>
<td>A parallel maximum sub-array algorithm for 1 dimensional arrays using an naive search algorithm.</td>
</tr>
<tr>
<td><strong>mst</strong></td>
<td>A parallel minimum spanning tree algorithm using Boruvka’s algorithm (also known as Sollin’s algorithm).</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions and Future Works

This thesis explores the idea of work-efficient schedulers. To achieve work efficiency, this work identifies elasticity as a key component in designing work efficient schedulers. This work presents two work-efficient elastic schedulers for 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 proposing a novel game theoretic model capable of considering parallel scheduling in both adaptive and dedicated environment. We show that the $\alpha|\beta$-elastic greedy scheduler, being a special 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 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 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 as well.

- 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 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 mediating the communication structure 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


